The 26th International Conference on Automated Planning and Scheduling

Proceedings of the 8th Workshop on

Heuristics and Search for Domain-independent Planning (HSDIP)

Edited by:

J. Benton, Daniel Bryce, Michael Katz, Nir Lipovetzky, Christian Muise, Miquel Ramırez, Alvaro Torralba

London, UK, 13/06/2016

Organizing Committee

J. Benton
NASA Ames Research Center and the RISE Foundation, USA

Daniel Bryce
SIFT, USA

Michael Katz
IBM Watson Health, Israel

Nir Lipovetzky
University of Melbourne, Australia

Christian Muise
MIT, USA

Miquel Ramrez
ANU, Australia

Álvaro Torralba
Saarland University, Germany

Foreword

Heuristic search is one of the main approaches in many domain-independent planning variants, including classical planning, temporal planning, planning under uncertainty and adversarial planning.

The workshop on Heuristics and Search for Domain-Independent Planning (HSDIP) is the eighth workshop in a series that started with the Heuristics for Domain-Independent Planning (HDIP) workshops at ICAPS 2007, 2009 and 2011. At ICAPS 2012, the workshop was held for the fourth time and was changed to its current name and scope to explicitly encourage work on search for domain-independent planning. It was very successful under both names. Many ideas presented at these workshops have led to contributions at major conferences and pushed the frontier of research on heuristic planning in several directions, both theoretically and practically. The workshops, as well as work on heuristic search that has been published since then, have also shown that there are many exciting open research opportunities in this area. Given the considerable success of the past workshops, and since it has de facto become an annual event, we intend to continue holding it annually.

The main focus of the HSDIP workshop series is on contributions that help us find a better understanding of the ideas and principles underlying current heuristics and search techniques, their limitations, ways for overcoming those limitations, as well as the synergy between heuristics and search. While the workshop series has originated mainly in classical planning, it is very much open to new ideas on heuristic schemes for more general settings, such as temporal planning, planning under uncertainty and adversarial planning. Contributions do not have to show that a new heuristic or search algorithm "beats the competition". Above all we seek crisp and meaningful ideas and understanding. Also, rather than merely being interested in the "largest" problems that current heuristic search planners can solve, we are equally interested in the simplest problems that they cannot actually solve well.

We hope that the workshop will constitute one more step towards a better understanding of the ideas underlying current heuristics, of their limitations, and of ways for overcoming those.

We thank the authors for their submissions and for their hard work.

June 2016 J. Benton, Daniel Bryce, Michael Katz, Nir Lipovetzky, Christian Muise, Miquel Ramirez, Álvaro Torralba.

Table of Contents

Evaluation of a Simple, Window-based, Replanning Approach to Plan Optimization
Shoma Endo, Masataro Asai and Alex Fukunaga **5**

Correlation Complexity of Classical Planning Domains
Jendrik Seipp, Florian Pommerening, Gabriele Röger and Malte Helmert **12**

Duality in STRIPS planning
Martin Suda **21**

Improving Performance by Reformulating PDDL into a Bagged Representation
Pat Riddle, Jordan Douglas, Mike Barley, and Santiago Franco **28**

On State-Dominance Criteria in Fork-Decoupled Search
Álvaro Torralba, Daniel Gnad, Patrick Dubbert and Jörg Hoffmann **37**

Decoupled Strong Stubborn Sets
Daniel Gnad, Martin Wehrle and Jörg Hoffmann **45**

Optimal Solitaire Game Solutions using A* Search and Deadlock Analysis
Gerald Paul and Malte Helmert **52**

Lifting Delete Relaxation Heuristics To Successor Generator Planning
Michael Katz, Dany Moshkovich and Erez Karpas **61**

Non-Deterministic Planning with Temporally Extended Goals: Completing the story for finite and infinite LTL
Alberto Camacho, Eleni Triantafillou, Christian Muise, Jorge Baier and Sheila McIlraith **68**

Abstraction Heuristics for Symbolic Bidirectional Search
Álvaro Torralba, Carlos Linares López and Daniel Borrajo **77**

Blind Search for Atari-like Online Planning Revisited
Alexander Shleyfman, Alexander Tuisov and Carmel Domshlak **85**

Delete-free Reachability Analysis for Temporal and Hierarchical Planning
Arthur Bit-Monnot, David Smith and Minh Do **93**

Cost-Optimal Algorithms for Hierarchical Goal Network Planning: A Preliminary Report
Vikas Shivashankar, Ron Alford, Mark Roberts and David Aha **102**

Monte Carlo Tree Search as a Hyper-heuristic Framework for Classical Planning
Otakar Trunda **111**

Evaluation of a Simple, Window-based, Replanning Approach to Plan Optimization

Shoma Endo, Masataro Asai, Alex Fukunaga
Graduate School of Arts and Sciences
The University of Tokyo

Abstract

The task of satisficing planning is to find the highest quality plan within a given time budget. Several postprocessing techniques for plan optimization techniques have been developed for improving the quality of a satisficing plan. In this paper, we first survey a class of plan optimization techniques which apply replanning to windows (subplans) of a given plan. We propose and evaluate a CH-WIN, a new window-based plan optimization algorithm based on AIRS, a previous technique for prioritizing plan windows according to discrepancies between actual and heuristic cost accruals within a window.

1 Introduction

Minimizing the cost of the solution is a key element in satisficing planning in order to obtain a reasonable solutions for practical real-world applications. There are two major approaches to addressing this problem. The first approach is the development of anytime search algorithms that initially generate some solution relatively quickly, then continue searching for better solutions in the remaining time available. Examples of this approach include Restarting WA* (Richter and Westphal 2010), AEES (Thayer, Benton, and Helmert 2012), Diverse Anytime Search (Xie, Valenzano, and Müller 2013) and so forth. Another line of work includes plan optimization techniques, such as Action Elimination (Nakhost and Müller 2010), Action Dependency (Chrpa, McCluskey, and Osborne 2012), Plan Neighborhood Graph Search(PNGS) (Nakhost and Müller 2010), Block Deordering (Chrpa and Siddiqui 2015). Although these approaches share the same objective, the key difference between anytime search algorithms and postprocessing algorithms is that the former can be applied to generate a plan from scratch, whereas the latter assumes that at least one satisficing plan is available as an input.

This paper focuses on plan optimization techniques. In particular, we investigate a class of techniques we call "window-based plan optimization". In this approach, some part of an existing plan (a "window") is selected, and a replanning algorithm is applied to this window in order to locally optimize the selected subplan. This process is repeated until time runs out.

We first evaluate R-WIN, a simple, baseline strategy which randomly selects the windows to be optimized. We then propose CH-WIN, an improved variation of AIRS, a previous method for plan optimization which was evaluated on domain-specific solvers for the 15-puzzle and a grid pathfinding domain AIRS (Estrem and Krebsbach 2012). CH-WIN uses the same criteria as AIRS for ranking candidate replanning windows, but uses an improved method for window selection. We show that CH-WIN outperforms previous algorithms including AIRS and PNGS.

Finally, we evaluate the performance of these optimizers when combined in sequence. Since the input of an optimization algorithm is a plan and its output is also a plan, we can apply a chain of several optimization algorithms. Although this requires additional optimization time, we empirically show that spending more time on optimization actually pays off, compared to the performance of a single plan optimization technique.

This paper is structured as follows. Section 2 reviews several existing plan-optimization frameworks. Section 3 propose a class of window-based plan optimization and its two instances, and evaluate their performance against existing framework. Section 4 investigates the performance of the sequence of several plan-optimization frameworks. We finally conclude with future remarks.

2 Background

A common strategy for plan optimization in sequential satisficing planning is to run an anytime search algorithm with some time limit, then allocate the remaining time to the postprocessing optimizer. The postprocessing phase can be iterated, so that it continues to improve the plan quality.

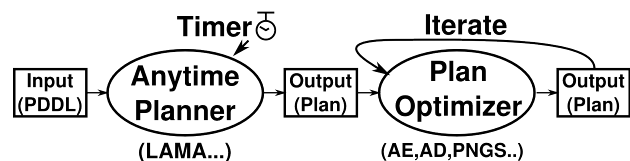


Figure 1: The concept of anytime planning combined with post-processing plan optimization.

Some algorithms run in polynomial-time while the others are based on search algorithms that may consume exponential amount of time. We call the first variants as the *polytime* optimization algorithms while the latter as the *search-based* optimization algorithms.

Action Elimination [**AE**] (Nakhost and Müller 2010) is a polytime optimization algorithm which iterates over the actions in the plan looking for unnecessary actions whose removal does not make the rest of the plan infeasible. A straightforward implementation runs in $O(pn^2)$ where n is the length of the plan and p is the maximum number of preconditions among actions in the plan.

Action Dependency analysis [**AD**](Chrpa, McCluskey, and Osborne 2012) is also a polytime optimization algorithm. It consists of two major steps. The first step uses a notion of *direct dependency* between a pair of actions and its transitive extension called *action dependency*. It removes all actions that is not dependent on the goal action, a pseudo action whose precondition is same as the goal condition. In the second step, AD uses the notion of *inverse action* which represents a pair of actions with effects that cancels each other. It removes a pair of inverse actions from the plan when the actions surrounded by the pair are not affected by the removal.

The first search-based optimization algorithm we are aware of is by Ratner and Pohl, who proposed a plan optimization algorithm which divides a suboptimal plan in to segments of length d, and tries to replace each segment with a better subplan obtained by applying A^* to locally optimize the segment (Ratner and Pohl 1986)

Plan Neighborhood Graph Search [**PNGS**] (Nakhost and Müller 2010) is a search-based plan optimization technique. It takes two parameters A and N, where A is a search algorithm and N is an integer. PNGS generates a *Plan Neighborhood Graph* of the plan as follows: Given a plan π and initial state s_0, consider the plan path $P = s_0, s1, ..., s_g$, which is the sequence of states that the system transitions through when each step of π is applied in sequence. For each such state s_i, it runs A until it generates N nodes. The resulting set of nodes, which includes the original solution itself, is then searched by another algorithm M', which is brute-force Dijkstra's algorithm in their experiments. When the refinement finishes, it doubles the number of nodes N and reapplies the algorithm.

Balyo, Barták, and Surynek investigated a window based, replanning approach for temporal planning which seeks to optimize makespan by applying a SAT-based planner to random or fixed sized windows of a suboptimal plan (Balyo, Barták, and Surynek 2012).

Block-deordering is a recent approach in which a totally ordered plan is first deordered and decomposed into a set of blocks (partially ordered plans), where each block is a set of steps that must not be interleaved with steps that are not in the block. Windows for planning are extracted based on these blocks, and a large-neighborhood search algorithm is applied in order to optimize the subplans (Siddiqui and Haslum 2013; 2015).

Finally, **Anytime Iterative Refinement of a Solution** (Estrem and Krebsbach 2012, AIRS) is a search-based plan optimization technique which runs a local search on a part of the plan. The portion is selected based on *Greedy Plateaus*, i.e. the parts of the plan where the heuristic estimate does not change as much as they should (relative to the actual costs). For every pair of states s_i, s_j in a plan, it calculates the ratio of the heuristic distance $h(s_i, s_j)$ to the actual cost $c(s_i, s_j)$ of the path between two states. In each iteration, the portion of the plan starting from s_i ending at s_j with the minimum ratio $\frac{h(s_i,s_j)}{c(s_i,s_j)-c_{min}}$ is selected, where c_{min} is the smallest action cost in the domain, and start the local search from s_i to s_j, using an admissible bidirectional search. It also has a mechanism to avoid searching the same region many times by storing the information about the portion where the admissible search failed to improve the plan.

The intuition behind AIRS is as follows. When the heuristic value is small while the actual cost is large between two states, then it would be highly likely that the greedy search algorithm which generated the initial plan has failed to find a good solution, and the better plan can be quickly found using admissible search because the optimal cost would be small according to the heuristic value.

3 Window-based Plan Optimization

However, the assumptions behind AIRS may not necessarily hold true for two reasons. First, it assumes that the value $h(s_i, s_j)$ is reasonably accurate, and that $c^*(s_i, s_j)$, the optimal cost of moving from s_i to s_j, is somewhat close to to $h(s_i, s_j)$. This is not necessarily true when, for example, h fails to correctly capture the search space structure and the difference between $c(s_i, s_j)$ and c^* is very small (or even zero). In such cases, attempting to replanning the given region may not be worthwhile (because the possible gain $c-c^*$ is small) or even futile. Second, due to the same reason, c^* may be too large for the admissible replanning algorithm to find the solution within a given time limit.

In order to separate the issues which are specific to AIRS from the general framework of local search on plan segments, we first define a class of optimization technique called *Window-based Plan Optimization*.

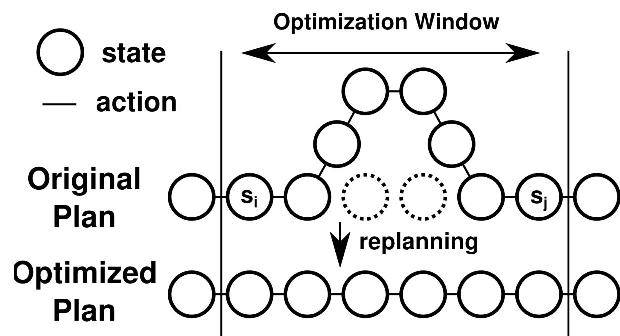


Figure 2: Simple figure describing the concept of window-based plan optimization.

In each iteration of plan refinement, Window-based optimization selects a *window* $W = (s_i, s_j)$ for the local search, which is a portion of the plan that is selected according to some criteria C, which is a free parameter of this framework. s_i, s_j are encoded as the initial state and the goal state of an input to some standard search algorithm A. If A successfully returns a better solution, it replace W with the new path. If A failed to find a better solution due to some criteria

such as the time limit, or because the resulting cost is worse than the original (note that A can be inadmissible), the solution is ignored. Window-based optimization is naturally an anytime algorithm which returns the current best solution any time it is interrupted.

A trivial baseline instance of this framework randomly selects a window of fixed size L. We call this variant **R-WIN** (Randomized Window-search). The parameter L controls the size of the window as well as the upper bound for the search conducted by the given algorithm A, so that it avoids solving the window that is beyond the capability.

The reason we adopted a randomized strategy rather than a deterministic strategy e.g. first select $W = (s_0, s_L)$, then $W = (s_1, s_{L+1})$ and so on, is to diversify the effort uniformly over the length of the plan. Otherwise, the effort put by the local search could be biased toward the beginning of the plan, while the possible refinements only exist near the end of the plan.

3.1 Empirical Evaluation of R-WIN

We evaluated R-WIN and AIRS on the International Planning Competition (IPC) benchmark instances included with the Fast Downward planner (Helmert 2006). We evaluated all methods on the IPC satisficing problem instances which are limited to STRIPS with action costs, since that is the subset of PDDL currently supported by our optimizer. All experiments in this paper were conducted on Xeon E5410@2.33GHz CPUs. We first solved these instances using Fast Downward LAMA2011 (Richter and Westphal 2010) using 15 minutes, 2GB setting. Among 958 problem instances, LAMA solved 828 problem instances in the first 15 minutes.

The resulting plans are then fed into the plan optimizers, under 2GB memory and the time limit of 15 minutes, resulting in 30 minutes of the total runtime. Since LAMA uses RWA* and produce multiple solutions, we selected the current best solution found when the time limit is reached. We evaluated the following optimizer configurations:

- LAMA (30 min) – this baseline continues to run LAMA for an additional 15 minutes (i.e., 30 minutes total) and returns the best solution found.
- Action Elimination (Nakhost and Müller 2010, AE)
- Action Dependency (Chrpa, McCluskey, and Osborne 2012, AD)
- Plan Neighborhood Graph Search (Nakhost and Müller 2010, PNGS): We used $N = 1000$ (as in the original paper).
- Anytime Iterative Refinement of a Solution (Estrem and Krebsbach 2012, AIRS)
- R-WIN: $L = 10$. Since R-WIN is a randomized algorithm, we ran the same experiments with two different random seeds in order to test the robustness of the algorithm. However, we could not observe any significant difference so we only show 1 set of results.

We implemented AE, AD, PNGS, and AIRS based on the descriptions in the original papers. However, for all search-based optimization algorithms (PNGS, AIRS, R-WIN), the replanning algorithm we used was Fast Downward with A* using the admissible LMcut heuristic (Helmert and Domshlak 2009). This differs from the original paper in that AIRS originally used Bidirectional A*(Pohl 1971), and PNGS used Dijkstra Search. This allows us to focus on the effect of the replanning strategy, as opposed to the underlying search algorithm.

Table 1 compares the ratio between the sum of the costs of the plans returned by LAMA with 15 minutes, against the results of various optimizers. Running LAMA for an additional 15 minutes results in very little improvement over the first 15 minutes, and is worse than all of the postprocessing optimizers that we evaluated. Although the difference was very small, R-WIN outperformed PNGS. This shows that a simple window-based algorithm like R-WIN can outperform a more complicated algorithms. Interestingly, AIRS was found to perform *worse* than R-WIN, a possible baseline of the Window-based optimization algorithms. Also, the polytime algorithms such as AE and AD perform reasonably good in terms of the runtime and the quality gain, even if these algorithms capture only the limited scope of optimization.

Algorithm	Harmonic Means of Ratios
LAMA(15min)	100%
LAMA(30min)	99.3%
AE	98.4%
AD	97.4%
AIRS	97.9%
PNGS	96.0%
R-WIN	**95.9%**

Table 1: Comparison of the cost-reduction ration between AE, AD, PNGS, AIRS, R-WIN, relative to bare LAMA with 15 minutes (100%). We took the harmonic means of the ratios over all problem instances. R-WIN outperformed PNGS, but with a small difference.

The complete domain-wise results in Table 2 (which also contains results for the next section) suggests that the performances of R-WIN and PNGS are significantly affected by the domain characteristics. For example, R-WIN performs better than PNGS in 12 domains out of 39 domains, although PNGS performs better than R-WIN in 18 domains. Thus, there is no overall dominance relationship between these optimization algorithms. We can also see that R-WIN fails to improve any plan on some domains such as floortile-sat11, parcprinter-08/sat11, pipesworld (both tankage and notankage versions), sokoban-sat08/11, woodworking-sat08. In these domains, we observed that R-WIN is consuming too much time on "difficult" window replanning problem instances. The same symptom was also observed in AIRS. This suggests that addressing these issues would result in a simple yet effective optimization algorithm.

3.2 CH-WIN

Since the results in the previous section suggested some possibilities for improvement within the Window-based opti-

domain	LAMA 30min.	AD	AE	AIRS	PNGS	R-WIN	CH-WIN
airport	99.9	100	100	100	100	100	**99.3**
barman11	99.9	94.8	91.0	100	100	**87.6**	96.0
blocks	100	100	87.8	91.1	80.1	84.9	**78.5**
depot	98.7	92.2	93.6	99.3	93.0	89.6	**88.7**
driverlog	99.0	97.1	96.4	98.7	93.7	92.2	**90.2**
elevators08	99.8	92.0	91.7	99.9	98.0	**86.3**	89.5
elevators11	99.9	92.7	**92.4**	100	100	95.5	97.3
floortile11	91.9	97.5	94.7	100	98.3	96.7	**80.4**
freecell	99.3	99.1	99.0	99.8	91.8	99.8	**95.1**
grid	100	100	100	100	**91.5**	92.7	91.9
logistics00	100	98.7	98.7	96.7	98.0	97.3	**94.8**
logistics98	100	99.0	98.8	99.2	99.6	98.5	**98.1**
miconic	100	100	100	89.7	89.7	90.0	**88.4**
nomystery11	100	100	100	100	100	100	**99.8**
openstacks08	**98.7**	100	100	100	100	100	100
openstacks11	**97.8**	100	100	100	100	100	100
parcprinter08	99.0	100	98.3	100	100	100	**94.2**
parcprinter11	99.0	100	100	100	100	100	**93.1**
parking11	99.8	99.5	99.5	100	100	99.3	**98.7**
pegsol08	97.1	100	100	100	100	94.9	**89.4**
pegsol11	91.7	100	100	100	100	96.0	**85.9**
pipesworld-not	97.6	99.8	93.1	98.3	93.9	100	**89.3**
pipesworld-t	97.3	100	95.3	98.6	92.8	100	**92.3**
rovers	99.9	96.8	**96.8**	99.7	99.4	98.3	98.9
satellite	99.9	99.1	98.9	97.2	98.1	**95.5**	96.6
scanalyzer08	99.3	100	100	100	98.6	96.2	**93.2**
scanalyzer11	99.3	100	100	100	98.1	96.2	**93.2**
sokoban08	95.2	100	98.4	100	93.9	99.7	**90.7**
sokoban11	94.6	100	98.2	100	93.2	100	**90.1**
tpp	99.7	99.5	99.5	98.0	98.8	95.4	**92.6**
transport08	100	94.2	**94.1**	100	99.3	95.9	97.1
transport11	99.8	92.3	**91.9**	100	100	97.5	99.6
trucks	**99.1**	100	100	100	100	100	100
visitall11	99.9	100	99.7	100	100	99.6	**99.5**
woodworking08	100	99.6	99.3	100	100	100	**88.2**
woodworking11	100	99.8	98.8	100	100	100	**90.8**
zenotravel	100	100	100	100	98.7	100	**97.3**
mean(harmonic)	99.3	98.4	97.4	97.9	*96.0*	*95.9*	**93.3**

Table 2: Comparison of the cost-reduction ratios between AE, AD, PNGS, AIRS, R-WIN and CH-WIN, relative to plain LAMA with 15 minutes (100%). The costs are the sum over all instances solved by LAMA within 15 minutes.

mization framework, we implemented a more sophisticated window selection scheme named **CH-WIN**. The name reflects the fact that it uses the ratio between the actual cost c and the heuristic value h, similar to AIRS (Estrem and Krebsbach 2012), but with several simple improvements that address the problems present in AIRS and R-WIN.

First, we improved the window selection / penalization criteria in AIRS. Although AIRS adopts a mechanism which gives a penalty to a window whose region overlaps with the previously selected windows and tries to avoid solving them, we found that the mechanism does not sufficiently deter the optimizer from duplicated work, sometimes repeatedly selecting the nearby windows. In CH-WIN, we never select the same segment twice. Windows are simply sorted into a priority queue according to the value of $h(s_i, s_j)/c(s_i, s_j)$, and no window is reinserted into the queue after being optimized. Thus CH-WIN simplifies AIRS by removing the penalty mechanism – there is no need to consider the overlaps between the currently selected window and the past windows whose replanning attempt has failed.

In fact, although it is not worthwhile to select the same segment more than twice , it is still possible to improve the segments which overlap it. There are two possible reasons that the previous attempt has failed. (1) First, the failure may be due to the large $c^*(s_i, s_j)$ which makes it too difficult for admissible algorithms to obtain a refined solution. Thus, replanning its shorter subsegments like (s_k, s_l) s.t. $i < k < l < j$ may succeed. (2) Second, the failure could be due to a successful replanning with no improvements, i.e. $c(s_i, s_j) = c^*(s_i, s_j)$. It suggests that its subsegment can be solved much quicker, which means that replanning is not so harmful. Also, it does not preclude the chance of improving a segment which overlaps it, such as (s_k, s_l) s.t. $k < i < l < j$. Thus, we consider the simplified criteria offers a wider and safer opportunity to improve the plan quality overall. As an implementation detail, each h, c, s_i and s_j is updated after the successful replanning of a window, in order to reflect the changes to the plan and the positions of states in the plan.

Second, we applied a time limit of 3 minutes for each replanning episode. This avoids the problem we observed in AIRS and R-WIN that some individual replanning attempts can consume too much time. It is possible that AIRS does not address this issue because in the original paper, AIRS was tested on problems where the replanning attempts were all relatively easy (under 0.1 second per episode in the 15 puzzle with domain-specific heuristics and grid pathfinding).

Third, along the same line as our second improvement above, we apply a maximum length to the replanning window, which is dynamically adjusted as the replanning succeeds or fails. The dynamic process has two parameters L and $\bar{L}$, and runs a binary search of the appropriate length as shown in Algorithm 1. The value of L trivially converges to some value L_∞.

We tested CH-WIN using the same experimental setting as in the previous section. The results are shown in Table 3, which shows that CH-WIN outperforms both R-WIN and PNGS algorithm. Per-domain results in Table 2 also

Algorithm 1 Binary search of L, used for adapting the size of the replanning window.

1: Initially $L = n/4$ and $\bar{L} = n$, where n is the length of the plan.
2: When CH-WIN selects the next window, it skips windows whose lengths exceed L. Skipped windows are inserted back to the end of the queue, so that future increases in L allow the optimizers to reconsider those candidates.
3: **if** replanning succeeds within the time limit, **then**
4: $L \leftarrow (L + \bar{L})/2$.
5: **else**
6: $L \leftarrow L/2$ and $\bar{L} \leftarrow L$.
7: **end if**

show that CH-WIN outperforms other algorithms in many domains (26 domains out of 36 domains).

Algorithm	Harmonic Means of Ratios
LAMA(15min)	100%
LAMA(30min)	99.3%
PNGS	96.0%
R-WIN	95.9%
CH-WIN	**93.3%**

Table 3: Summary of the comparison of the cost-reduction ration between PNGS, R-WIN and CH-WIN relative to bare LAMA with 15 minutes (100%). We took the harmonic means of the ratios over all problem instances. CH-WIN outperformed both PNGS and R-WIN.

4 Evaluating Sequences of Optimization Techniques

As a natural consequence of Plan Optimizer taking a plan as an input and emits a plan as an output, we can combine several different optimizer algorithms in sequence, hoping that each algorithm addresses different kind of inefficiencies / redundancy in the suboptimal plan.

This idea is not new. Aras (Nakhost and Müller 2010) applies Action Elimination to the input, then run PNGS to its output. Siddiqui and Haslum also applied BDPO2 to the results of PNGS in one of their experiments (Siddiqui and Haslum 2015). However, currently there are few in-depth analysis on the effect of combining many postprocessing optimization algorithms.

4.1 Sequences of Poly-Time Optimizers

We first evaluated two polytime algorithms (AD, AE) applied in different orders (AD+AE and AE+AD) and in iterations (AE+AD+AE, AE+AD+AE+AD, ...) to ensure that simply sequencing these polytime optimizers does not further improve the plan quality.

As shown in Table 4, repeatedly alternating between AE and AD does not improve the plan quality. Although there is a slight difference in the result depending on which optimizer is applied first, it happens only in the depot and elevators08 domains. This yields two observations: First, when we use AE and AD as a part of optimizer sequence, applying each of AE and AD once is enough to remove the inefficiency addressed by AE and AD. Second, we need to add either a search-based optimizer (such as PNGS, AIRS, CH-WIN) or a new polytime optimizer (future work) to the optimizer sequence in order to get a better plan.

domain	AD	AD AE	AD AE AD	AD AE AD AE	AE	AE AD	AE AD AE	AE AD AE AD
mean(harmonic)	98.4	97.4	97.4	97.4	97.4	97.4	97.4	97.4

Table 4: Result of applying poly-time optimizers iteratively to plans obtained by 15-minute runs of LAMA. The harmonic means of the ratios over all instances are shown.

4.2 Sequences Poly-time Optimizers and a Search-Based Optimizer

We next evaluated two polytime algorithms (AD, AE) applied in this order (AD+AE) with one of three search-based algorithms (AIRS, PNGS, R-WIN, CH-WIN), i.e., AD + AE + S configuration where S is one of AIRS, PNGS, R-WIN, CH-WIN.

We tested these configurations with the same resource limitations (time, memory) as in the previous experiments. Each optimizer configuration processes the input plan with a 15 minutes time limit (total, shared among all components of the optimizer shared) and 2GB memory limit. The input plans are the best results output by running LAMA for 15 minutes, with a 2GB limit on each problem instance. Polytime algorithms usually finish very quickly, and their runtimes are negligible compared to the search-based optimizers. Thus, we assume most of the 15 minutes of runtime for postprocessing is consumed by the search-based optimizers in the configuration.

As shown in the previous section, repeating AD and AE steps do not improve the quality nor result in different plans, so we do not test configurations which repeat several AD+AE sequences (such as AE+ AD+ AE+ AD+ PNGS).

Table 5 shows the results of running these configurations of multiple optimizers. Combining several optimizers behaved as expected: The results of search-based optimizers with polytime optimizers tends to be better than without them, and polytime optimizers themselves are also helped by search-based optimizers. We can see that the best performer was AE + AD + CH-WIN, a configuration which applies Action Elimination first, then applies the Action Dependency to the output of AE, then applies CH-WIN to the output of AD.

We also evaluated configurations where each of above configuration is followed by an additional applications of AE and AD (i.e. AE+ AD+S+ AE+ AD), but we could not observe any quality improvement: The harmonic-mean scores were 95.6, 94.4, 94.0, 91.8 (for S=AIRS, PNGS, R-WIN, CH-WIN, respectively), which means that the search-based optimizers we used in this paper tend not to regener-

ate the inefficiency which can be detected by the polytime optimizers. This might be a natural consequence of the underlying replanner A being an admissible (optimal) planner. Investigating the cases which use inadmissible planner as A is future work.

4.3 Sequences With Multiple Search-Based Optimizers

We finally investigated the performance of the combinations of multiple search-based optimizers (PNGS, R-WIN, CH-WIN). We focus on the synergy that can happen on several different optimizer algorithms because there are still some domains in Table 5 where PNGS and R-WIN yield better results than CH-WIN.

We do not include AIRS in this experiments because CH-WIN is an improved version of AIRS and we showed in previous sections that CH-WIN outperforms AIRS. Also, we do not need to interleave AE+AD between the runs of search-based optimizers because we showed, in the previous section, that the results emitted by the search-based optimizers usually do not contain inefficiency which can be detected by poly-time optimizers. Thus, for each search-based optimizer S_1 and S_2, we tested AE+ AD+ S_1+ S_2 (where S_1 and S_2 should be different optimizers).

In this experiment, we run AE and AD first, then S_1 for 7.5 minutes, then S_2 for 7.5 minutes. In all cases, the replanning is constrained under 2GB memory limitation, and the input was again the results of LAMA being run for 15 minutes, 2GB constraints.

The results in Table 6 show that the combination of different search-based optimizers does *not* yield better results than AE + AD + CH-WIN. This supports our claim that CH-WIN has both a proper window selection scheme (use of c/h ratio), which is lacking in R-WIN and PNGS, and a window scaling scheme (adaptive window size), which is lacking in R-WIN and AIRS, thus outperforming those algorithms by successfully focusing the replanning effort on to the region of "most room for improvements".

5 Conclusion

We proposed and evaluated a simple, window-based replanning approach to plan optimization. Although it has been conjectured that a sliding window based techniques is "unlikely to find relevant subproblems in general planning problems where the sequential plan is often an arbitrary, interleaving of separate threads" (Siddiqui and Haslum 2013), our results indicate that a window based approach is sufficient to obtain significant plan quality improvements compared to the baseline, as well as to previous methods such as PNGS. A comparison with the block-deordering strategies in BDPO2 (as well as its combinations with other algorithms) is an avenue for future work (Siddiqui and Haslum 2013).

One advantage of our approach is simplicity of implementation. CH-WIN is a pure, wrapper-based approach which uses a standard off-the-shelf planner for replanning, as well as the heuristic value computations needed for window selection. Our implementation of CH-WIN required no modi-

domain	AE AD AIRS	AE AD PNGS	AE AD R-WIN	AE AD CH-WIN
airport	100	100	100	**99.3**
barman11	91.0	91.0	85.0	**89.2**
blocks	84.2	82.9	83.9	**80.1**
depot	93.4	89.9	**87.4**	88.0
driverlog	95.6	92.3	89.6	**87.8**
elevators08	91.6	90.8	**80.5**	82.5
elevators11	92.4	92.4	**87.7**	89.8
floortile11	94.7	93.2	93.7	**78.9**
freecell	98.9	**91.2**	99.0	94.6
grid	100	**91.5**	92.7	91.9
logistics00	96.2	97.6	96.8	**93.4**
logistics98	98.3	98.5	97.4	**97.1**
miconic	89.8	89.8	89.8	**88.4**
nomystery11	100	100	100	**99.8**
openstacks08	100	100	100	100
openstacks11	100	100	100	100
parcprinter08	98.3	98.3	98.3	**94.2**
parcprinter11	100	100	100	**93.1**
parking11	99.5	99.5	98.2	**98.6**
pegsol08	100	100	94.9	**89.4**
pegsol11	100	100	92.1	**85.9**
pipesworld-not	92.6	91.6	93.1	**87.8**
pipesworld-t	93.9	90.5	95.3	**90.4**
rovers	96.7	96.8	**96.3**	96.4
satellite	96.7	97.3	**95.6**	**95.6**
scanalyzer08	100	98.6	96.4	**93.2**
scanalyzer11	100	98.5	98.2	**93.2**
sokoban08	98.4	94.4	98.1	**88.2**
sokoban11	98.2	93.7	98.2	**87.2**
tpp	97.6	98.8	93.0	**92.5**
transport08	94.1	93.1	91.1	**92.1**
transport11	91.9	91.9	**90.9**	91.8
trucks	100	100	100	100
visitall11	99.7	99.7	**99.3**	**99.3**
woodworking08	99.3	99.3	99.3	**87.6**
woodworking11	98.8	98.8	98.8	**88.9**
zenotravel	100	100	100	100
mean(harmonic)	95.6	94.4	94.0	**91.8**

Table 5: Results of using poly-time optimizers and search-based optimizers, on the plans obtained by 15 minutes runs of LAMA. The harmonic means of the ratios over all problem instances are shown. The poly-time optimizers did not harm the successive application of search-based optimizers, and the combination using our CH-WIN was the best performer, same as in the previous experiments.

	AE AD					
S_1	PNGS		R-WIN		CH-WIN	
S_2	R-WIN	CH-WIN	PNGS	CH-WIN	PNGS	R-WIN
airport	100	99.6	100	99.5	99.6	99.6
barman11	86.2	90.1	85.9	85.6	89.9	85.8
blocks	81.9	80.1	85.2	78.6	79.9	79.6
depot	88.4	88.3	87.8	88.5	88.1	87.6
driverlog	89.8	87.4	89.9	86.9	87.0	86.4
elevators08	86.5	86.3	80.0	81.2	84.1	80.3
elevators11	91.4	92.2	88.7	88.5	90.9	88.3
floortile11	90.8	78.6	86.9	78.6	78.9	78.9
freecell	91.7	91.4	98.2	98.8	91.7	95.8
grid	91.5	91.5	91.5	92.3	91.5	91.9
logistics00	96.5	94.5	96.6	94.1	94.2	94.2
logistics98	97.6	97.6	97.9	97.6	97.6	96.8
miconic	89.4	87.3	89.6	87.7	88.6	88.6
nomystery11	100	100	100	100	100	100
openstacks08	100	100	100	100	100	100
openstacks11	100	100	100	100	100	100
parcprinter08	97.5	94.5	97.8	94.2	94.2	94.2
parcprinter11	98.5	94.2	99.5	93.7	93.2	93.2
parking11	98.6	98.9	98.1	98.7	98.9	98.2
pegsol08	93.2	88.3	91.4	87.1	91.4	93.2
pegsol11	87.7	85.9	87.7	83.8	87.7	89.4
pipesworld-not	91.6	88.8	91.6	88.7	88.5	88.7
pipesworld-t	91.0	90.1	90.8	91.2	89.0	91.4
rovers	96.7	96.4	96.5	96.2	96.4	96.3
satellite	95.2	96.0	95.2	94.9	95.9	94.8
scanalyzer08	96.6	94.8	96.1	94.2	94.0	94.0
scanalyzer11	96.7	95.6	96.2	95.7	94.8	94.3
sokoban08	95.1	89.0	96.5	94.0	91.2	91.2
sokoban11	94.4	88.7	94.6	91.6	90.6	90.4
tpp	95.1	94.7	94.7	92.9	93.7	91.7
transport08	90.9	91.7	90.9	91.8	91.7	91.6
transport11	90.9	91.4	91.0	91.0	90.9	91.1
trucks	100	100	100	100	100	100
visitall11	99.4	99.3	99.4	99.0	99.4	99.5
woodworking08	98.9	97.6	98.9	89.0	89.5	89.0
woodworking11	98.8	98.8	98.8	92.4	92.5	92.4
zenotravel	100	100	100	100	100	100
mean(harmonic)	93.4	92.5	93.6	92.2	92.1	92.0

Table 6: Results of applying polytime optimizers (AE and AD) and multiple search-based optimizers (S_1 and S_2) in sequence, on the plans obtained by 15 minutes runs of LAMA. The numbers show the harmonic means of the ratios over all problem instances. Each search-based optimizer is run for 7.5 minutes under 2GB constraints.

Detailed analysis on the log file has shown that the poly-time optimizer failed to find a better plan within the time limit in many instances and simply passed the input plan to the next optimizer.

Overall, participation of CH-WIN resulted in better plans compared to those without CH-WIN, and none of these configuration outperformed the configuration AE + AD + CH-WIN (reduction ratio 91.8%), indicating that CH-WIN dominates AIRS, R-WIN, PNGS.

fications to the Fast Downward planner. The CH-WIN code is simple: aside from the simple window selection schemes described in this paper, the only thing that the CH-WIN code needs to do is to perform a forward simulation of a PDDL problem (for window start/end point construction), i.e., it is no more complicated than a simple plan validator. Thus, CH-WIN is a promising, "quick-and-dirty" approach for plan optimization.

6 Acknowledgments

This research was supported by a JSPS Grant-in-Aid for JSPS Fellows and a JSPS KAKENHI grant.

References

Balyo, T.; Barták, R.; and Surynek, P. 2012. Shortening Plans by Local Re-planning. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, 1022–1028.

Chrpa, L., and Siddiqui, F. 2015. Exploiting Block Deordering for Improving Planners Efficiency. In *Proc. IJCAI*.

Chrpa, L.; McCluskey, T. L.; and Osborne, H. 2012. Optimizing Plans through Analysis of Action Dependencies and Independencies. In *Proc. ICAPS*.

Estrem, S. J., and Krebsbach, K. D. 2012. AIRS: Anytime Iterative Refinement of a Solution. In *FLAIRS Conference*.

Helmert, M., and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proc. ICAPS*.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR* 26:191–246.

Nakhost, H., and Müller, M. 2010. Action Elimination and Plan Neighborhood Graph Search: Two Algorithms for Plan Improvement. In *Proc. ICAPS*, 121–128.

Pohl, I. 1971. Bi-directional Search. *Machine Intelligence* 6:127–140.

Ratner, D., and Pohl, I. 1986. Joint and LPA*: Combination of Approximation and Search. In *Proceedings of the 5th National Conference on Artificial Intelligence. Philadelphia, PA, August 11-15, 1986. Volume 1: Science.*, 173–177.

Richter, S., and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *J. Artif. Intell. Res.(JAIR)* 39(1):127–177.

Siddiqui, F. H., and Haslum, P. 2013. Plan Quality Optimisation via Block Decomposition. In *Proc. IJCAI*.

Siddiqui, F. H., and Haslum, P. 2015. Continuing Plan Quality Optimisation. *J. Artif. Intell. Res.(JAIR)* 54:369–435.

Thayer, J. T.; Benton, J.; and Helmert, M. 2012. Better Parameter-Free Anytime Search by Minimizing Time Between Solutions. In *Proc. SoCS*, 120–128.

Xie, F.; Valenzano, R. A.; and Müller, M. 2013. Better Time Constrained Search via Randomization and Postprocessing. In *Proc. ICAPS*.

Correlation Complexity of Classical Planning Domains

Jendrik Seipp and **Florian Pommerening** and **Gabriele Röger** and **Malte Helmert**
University of Basel
Basel, Switzerland
{jendrik.seipp,florian.pommerening,gabriele.roeger,malte.helmert}@unibas.ch

Abstract

We analyze how complex a heuristic function must be to directly guide a state-space search algorithm towards the goal. As a case study, we examine functions that evaluate states with a weighted sum of state features. We measure the complexity of a domain by the complexity of the required features. We analyze conditions under which the search algorithm runs in polynomial time and show complexity results for several classical planning domains.

Introduction

Recently, *potential heuristics* (Pommerening et al. 2015) have been introduced as a new class of heuristics for classical planning. A potential heuristic is defined by specifying a numerical (possibly negative) *weight* for every fact of a planning task. The heuristic value of a state is then simply the sum of weights of the facts that are present in that state. Potential heuristics can be viewed as linear combinations of trivial *indicator functions*, where each indicator function tests whether a certain fact is present in the given state.

Due to their simple structure, potential heuristics can be evaluated very efficiently. Of course, the quality of their heuristic estimates critically depends on the choice of weights. In past work, finding suitable weights has been cast as an optimization problem with encouraging results (Pommerening et al. 2015; Seipp, Pommerening, and Helmert 2015).

However, it is clear that for challenging planning tasks, such simple potential heuristics cannot be truly informative, as complex interactions between different state variables cannot be adequately captured. Fortunately, the idea can be readily generalized by considering indicator functions for more complex state features than individual facts. An obvious generalization is to test for the presence of a *set* (or conjunction) of facts, similar to the generalization from the $h^{\max}$ heuristic to Haslum and Geffner's h^m heuristics (2000) or to the generalization from atomic to general projections in pattern database (PDB) heuristics [e.g., Edelkamp 2001].

It is easy to see that with such a generalization, arbitrary heuristics can be expressed as potential heuristics: in the extreme case, we can introduce a separate feature for every single state s and set its weight to the actual cost-to-goal $h^*(s)$ of that state. Again, this is analogous to the h^m heuristics, which converge to h^* as m increases to the number of facts of the planning task, and to PDB heuristics, which similarly converge to h^* as the set of pattern variables grows to include all variables. However, it is equally easy to see that in all three cases, the size of the representation explodes, and the heuristics become unmanageable on their way to perfection.

This raises the question how complex these heuristics need to become in order to faithfully capture the critical interactions between state variables. Many planning domains are known to admit polynomial domain-specific solution algorithms [e.g., Helmert 2003]. Perhaps "simple" heuristics only considering conjunctions of 2 or 3 facts are already highly accurate in these "simple" domains?

Unfortunately, there is bad news in the literature: Helmert and Mattmüller (2008) showed that h^m and (single) PDB heuristics based on conjunctions of bounded size give rise to arbitrarily bad heuristics in all domains they studied. However, they also showed that *additive* heuristics based on multiple PDBs can be significantly more accurate. This is not just good news for PDBs but also for potential heuristics, which are additive combinations of simpler heuristics by definition.

So just how complex does a potential heuristic have to be so that solving a planning task becomes simple? Following in Hoffmann's (2005) footsteps, we formalize this question by considering *per-domain* results for the *state space topology* of planning tasks. Hoffmann studied the search space topology of a fixed heuristic, namely the optimal delete relaxation heuristic h^+. Delete relaxation heuristics are rarely perfect but frequently good: to quantify this, Hoffmann focused on the size of local minima in the state space to distinguish "easy" from "difficult" domains for h^+.

In contrast, potential heuristics can be as accurate as we wish, at a cost in heuristic complexity. To reflect this degree of control, in our theoretical analysis we are more demanding with state space topology, looking for heuristics that exhibit *no local minima at all*. The question, then, is how complex – measured in the size of the conjunctions required – a potential heuristic needs to be in order to have no local minima.[1]

[1]Throughout the paper, by "local minimum" we mean any state which does not have a successor with lower heuristic value. This includes states within heuristic plateaus.

In this paper, we study this complexity measure for a number of well-known planning domains. It turns out that the results are very encouraging, motivating further study of potential heuristics with conjunctive features. We believe that this outcome is also relevant to researchers with no particular interest in potential heuristics, or even heuristic search.

At its core, the complexity measure we introduce describes how tightly interrelated different aspects of a planning task are, and to what extent these aspects can be considered separately. Within planning as heuristic search, such a measure is clearly relevant for approaches such as planning with pattern databases (Edelkamp 2001; Haslum et al. 2007; Pommerening, Röger, and Helmert 2013), critical-path heuristics (Haslum and Geffner 2000; Haslum, Bonet, and Geffner 2005), semi-relaxed plan heuristics (Keyder, Hoffmann, and Haslum 2014), conjunctive landmarks (Keyder, Richter, and Helmert 2010), or flow heuristics with merges (Bonet and van den Briel 2014). However, we think that such a measure of "interrelatedness" can be equally useful for non-heuristic planning approaches, such as factored planning (Brafman and Domshlak 2013), planning with decision diagrams [e.g., Torralba 2015], and compilations to SAT [e.g., Rintanen 2012; Suda 2014].

The general idea of measuring the degree of interrelatedness between state variables of a planning task is not new. In a line of research with very similar motivations to ours, Chen and Giménez (2007; 2009) studied several notions of *width* for planning tasks, where low width implies low complexity of planning. In the same spirit, Lipovetzky and Geffner (2012) also introduced a notion of *width* (different from those of Chen and Giménez) and exploited it to efficiently solve a large number of standard planning benchmarks. We return to this work towards the end of the paper, where we discuss the relationship between our complexity measure and the existing notions of width.

Planning Formalism

We consider SAS$^+$ (Bäckström and Nebel 1995) planning tasks $\Pi = \langle \mathcal{V}, \mathcal{O}, s_\mathrm{I}, s_\star \rangle$, where $\mathcal{V}$ is a finite set of state variables, $\mathcal{O}$ is a finite set of operators, s_I is the initial state, and $s_\star$ is the goal.

Each state variable $v \in \mathcal{V}$ has a finite domain $dom(v)$. A pair $\langle v, d \rangle$ with $v \in \mathcal{V}$ and $d \in dom(v)$ is called a *fact*. A set of facts is *consistent* if all contained facts belong to different variables. A consistent set of facts p is called a *partial variable assignment*. We write $vars(p)$ to denote the set of variables to which the facts in p belong. For $v \in vars(p)$ we write $p[v]$ to denote the value $d \in dom(v)$ for which $\langle v, d \rangle \in p$. If $vars(p) = \mathcal{V}$, p is called a *state*.

The initial state s_I is a state, and the goal $s_\star$ is a partial variable assignment. A state s is *consistent with* partial variable assignment p if $p \subseteq s$. A state s is a *goal state* if it is consistent with the goal $s_\star$. In some contexts, we refer to partial variable assignments as (state) *features* and say that a state *has the feature* F if it is consistent with F.

Each *operator* $o \in \mathcal{O}$ is given as a pair $o = \langle pre(o), eff(o) \rangle$, where the *precondition* $pre(o)$ and the *effect* $eff(o)$ are partial variable assignments. Operator o is *applicable* in state s if s is consistent with $pre(o)$. In this case, o may be *applied* in s, yielding the *successor state* $s[\![o]\!]$ defined by $s[\![o]\!][v] = eff(o)[v]$ for all $v \in vars(eff(o))$ and $s[\![o]\!][v] = s[v]$ for all other variables v. We write $\mathrm{succ}(s)$ for the set of all successor states of s, i.e., $\mathrm{succ}(s) = \{s[\![o]\!] \mid o \in \mathcal{O} \text{ is applicable in } s\}$. Our focus in this paper is on planning algorithms that do not provide quality guarantees for the plans they find, and hence we do not consider operator costs.

For a state s, an *s-plan* $\langle o_1, \dots, o_n \rangle$ is a finite sequence of operators such that $s[\![o_1]\!][\![o_2]\!] \dots [\![o_n]\!]$ is a goal state. We say that s is *solvable* if an s-plan exists and *unsolvable* otherwise. The task Π is solvable if the initial state s_I is solvable. A state s is *reachable* if $\langle \mathcal{V}, \mathcal{O}, s_\mathrm{I}, s \rangle$ is solvable. Finally, a *heuristic* is a function mapping states to $\mathbb{Z} \cup \{\infty\}$.

Potential Heuristics

Potential heuristics were introduced by Pommerening et al. (2015) as linear combinations of indicator functions, where each indicator function tests if a given fact is contained in the evaluated state. We generalize the definition to allow conjunctive state features. Throughout the paper, we write indicator functions using Iverson brackets (Knuth 1992).

Definition 1 (potential heuristic)**.** *Let* Π *be a planning task, let* $\mathcal{F}$ *be a set of state features of* Π*, and let* $w : \mathcal{F} \to \mathbb{Z} \cup \{\infty\}$.

The potential heuristic *with features* $\mathcal{F}$ *and weight function* w *is the function* φ *mapping each state* s *of* Π *to the integer*

$$\varphi(s) = \sum_{F \in \mathcal{F}} w(F)[F \subseteq s].$$

Note that we limit the definition to integer or infinite weights because these are sufficient for our purposes and simplify presentation. In other contexts, it may be preferable to permit arbitrary real-valued weights.

We measure the level of complexity of a potential heuristic by the size of the largest conjunction it uses as a feature, which we call its *dimension*.

Definition 2 (dimension)**.** *A potential function with features* $\mathcal{F}$ *has* dimension $\max_{F \in \mathcal{F}} |F|$.

Rephrasing what we said earlier using this terminology, previous work introduced potential heuristics of dimension 1, while we consider arbitrary dimensions.

The dimension of a potential heuristic is not the only natural way to measure its complexity. Alternative, more fine-grained measures include the number of features or the sum of features sizes. We choose to focus on the dimension because our results do not require more fine-grained measures and because dimension is a natural analogue to well-known parameters of other heuristics, such as the parameter m in the h^m heuristics and the pattern size in PDB heuristics.

For tasks with n state variables, potential heuristics of dimension d can be evaluated in time $O(n^d)$. In the common case where a family of planning tasks has a fixed bound on the number of effects in each operator, this can be improved to $O(n^{d-1})$ with incremental computations, i.e., when asked

to compute the heuristic value of a state given its parent state, generating operator and parent heuristic value. (To see this, note that if an operator changes k state variables, then only features involving at least one of these k state variables and hence at most $d-1$ *other* state variables need to be considered. The total number of such features can be bounded by $2^k \cdot (n-k)^{d-1}$, which is $O(n^{d-1})$ for constant k.) In particular, in this case potential heuristics of dimension 1 can be incrementally computed in constant time and potential heuristics of dimension 2 can be incrementally computed in time $O(n)$.

State Space Topology

We want to study potential heuristics without local minima. To formalize this, we must first clarify what we mean by having no local minima. A tentative definition might be: "every non-goal state has a successor with lower heuristic value". However, this is too strict: in a finite state space, such a definition implies that there is a strictly descending path towards a goal state from every state, which is impossible to satisfy if the task has any unsolvable states.

Hence, we only require that *solvable* states have successors with lower heuristic value. To avoid a heuristic search algorithm from getting trapped in an unsolvable region of the state space, we also require that unsolvable successors s' of a solvable state s never have a lower heuristic value than s.

A second problem is that planning tasks often include "impossible" states that violate physical constraints, such as two blocks being stacked on top of each other in the Blocksworld domain. It would be unnecessarily restrictive to require that the state space topology is also well-behaved for such impossible states. However, there is in general no simple way to distinguish possible from impossible states without complicating the definition of planning tasks. A simple remedy is to restrict attention to *reachable* states.

Definition 3 (alive)**.** *A state is* alive *if it is solvable, reachable, and not a goal state.*

We can now introduce two criteria that together imply absence of local minima.

Definition 4 (descending)**.** *A heuristic h is* descending *if all alive states have an improving successor. In symbols, for all states s:*

$$s \textit{ alive} \Longrightarrow \exists s' \in \text{succ}(s) : h(s') < h(s).$$

Definition 5 (avoiding dead ends)**.** *A heuristic h* avoids dead ends *if all improving successors of alive states are solvable. In symbols, for all states s and s':*

$$s \textit{ alive} \wedge s' \in \text{succ}(s) \wedge h(s') < h(s) \Longrightarrow s' \textit{ solvable}.$$

Given these two properties typical heuristic search algorithms for satisficing planning are guided directly towards the goal. We give a formal proof for simple hill-climbing (Algorithm 1).

Theorem 1. *Let h be a descending, dead-end avoiding heuristic for a planning task Π. Let $L = h(s_\mathrm{I}) - \min_{s \in S} h(s)$, where S is the set of all states of Π.*

Algorithm 1 Simple hill-climbing.

$s \leftarrow s_\mathrm{I}$
$\pi \leftarrow \langle\rangle$
while s is no goal state **do**
  improvement $\leftarrow$ **false**
  for $s' \in \text{succ}(s)$, in any order **do**
    if $h(s') < h(s)$ **then**
      improvement $\leftarrow$ **true**
      append $o \in \mathcal{O}$ with $s[\![o]\!] = s'$ to π
      $s \leftarrow s'$
      break
  if *improvement* is **false then**
    fail
return π

Then simple hill-climbing with h solves Π after at most L state expansions if Π is solvable and returns with failure after at most L state expansions if Π is unsolvable.

Proof: Consider the case where Π is solvable. For the while loop, we show the loop invariant that s is reachable and solvable. Reachability is trivial. For solvability, s is initially solvable, and in every iteration of the loop, the chosen state s' is solvable because s is alive (because it is not a goal state and due to the loop invariant, it is reachable and solvable), s' is an improving successor of s and h avoids dead ends.

We next show that the algorithm terminates by returning a plan (rather than failing or not terminating). Because h is descending, an improving state is always found inside the for loop, so the while loop never fails. Moreover, the while loop must finish with a bounded number of iterations because $h(s)$ decreases in every iteration and hence the sequence of expanded states never repeats. This proves that the algorithm terminates and also establishes the stated bound on L. (Note that $h(s)$ is an integer and hence must decrease by at least 1 in every iteration.)

In the case where Π is unsolvable, simple hill-climbing fails as soon as there is no more successor with lower heuristic value. As in the previous case, $h(s)$ cannot decrease more than L times, bounding the number of steps. □

The same result holds, with the same proof, for steepest ascent hill-climbing, a variant of hill-climbing that always moves to a successor s' minimizing the h value.

In the case where Π is solvable, the result also extends to the three most common satisficing planning algorithms: standard greedy best-first search a.k.a. eager greedy search (Russell and Norvig 2003), greedy best-first search with deferred evaluation a.k.a. lazy greedy search (Richter and Helmert 2009) and enforced hill-climbing (Hoffmann and Nebel 2001). To see this, observe that for descending, dead-end avoiding heuristics applied to solvable planning tasks, eager search expands the same states as steepest ascent hill-climbing, enforced hill-climbing expands the same states as simple hill-climbing, and lazy search evaluates the same states as simple hill-climbing.

We conclude this section by looking in a bit more depth at the requirement of avoiding dead ends. A special case in which this property holds for all heuristics are tasks

where no solvable states have unsolvable successors. Hoffmann (2005) calls such planning tasks *harmless*. A common special case of harmless planning tasks are *undirected* tasks, where $s \in \text{succ}(s')$ iff $s' \in \text{succ}(s)$.

Instead of heuristics that *avoid* dead ends, one can make the stricter requirement of *recognizing* dead ends (Hoffmann 2005), i.e., requiring $h(s) = \infty$ for all unsolvable states. This stricter property is not needed for Theorem 1, but if it is given and the heuristic is known to be *safe* (i.e., $h(s) = \infty$ guarantees that the state is unsolvable), then the equivalent of the theorem for enforced hill-climbing, eager greedy search and lazy greedy search also holds in the case of unsolvable planning tasks.

Instead of strengthening the requirement of avoiding dead ends, one could also consider the weaker requirement that unsolvable states are never among the best successors (minimizing h) of solvable states. This weaker requirement would still be sufficient for establishing a result like Theorem 1 for steepest ascent hill-climbing and eager greedy search, but not for simple or enforced hill-climbing or lazy greedy search.

Correlation Complexity

We now put the pieces of the previous two sections together: *correlation complexity* measures how complex a potential heuristic must be to obtain a favorable state space topology.

Definition 6 (correlation complexity of a planning task)**.** *The* correlation complexity *of a planning task* Π *is the minimum dimension* d *of all descending, dead-end avoiding potential heuristics for* Π.

The correlation complexity of a planning task is trivially bounded from above by the number of state variables n: in the worst case, we can define a feature with weight $h^*(s)$ for every state s, and because $|s| = n$, such a potential heuristic has dimension n. In particular, this guarantees that the correlation complexity of planning tasks is well-defined.

The definition can be extended to *planning domains*, which for the purposes of this paper are simply (usually infinite) sets of planning tasks.

Definition 7 (correlation complexity of a planning domain)**.** *The* correlation complexity *of a planning domain is the maximal correlation complexity of all planning tasks in the domain, or* ∞ *if no maximum exists.*

If a domain has low correlation complexity, this is a sign that no complex interactions between variables need to be considered in order to solve planning tasks in this domain. Hence, low correlation complexity is an indication that a planning domain is "easy".

A formal tractability result for planning in such a domain does not immediately follow because Definition 7 does not guarantee that a low-dimension potential heuristic for a given planning task is easy to construct – it only guarantees that such a potential heuristic exists. Moreover, planning domains with low correlation complexity can have exponentially long plans. For example, it is easy to construct "binary counter" tasks $(\Theta_i)_{i\geq 1}$ with correlation complexity 1 where Θ_i requires plans of length 2^i to solve. In the absence of such complications, low correlation complexity indeed implies tractability.

Theorem 2. *Let* $\mathcal{D}$ *be a planning domain with correlation complexity* $d < \infty$*, and let* p *be a polynomial such that given* $\Pi \in \mathcal{D}$ *with encoding size* n*,*

1. *a descending, dead-end avoiding potential heuristic* φ_Π *of dimension* d *can be computed in time* $p(n)$*, and*
2. *feature weights are polynomially bounded:* $|w(F)| \leq p(n)$ *for all features* F *of* φ_Π*.*

Then plan generation in $\mathcal{D}$ *can be solved in polynomial time.*

Proof: A task with encoding size n has at most n state variables, and hence φ_Π has no more than $O(n^d)$ features. Together with the bound on the individual weights, it follows that $|\varphi_\Pi(s)| \leq O(n^d)p(n)$ for all states s, and hence the difference between the heuristic values of any two states is bounded by a polynomial in n.

The result follows with Theorem 1, as L is bounded by a polynomial in n, and each heuristic evaluation can be performed in time $O(n^d)$, which is also polynomial in n. □

Properties of Potential Heuristics

In the rest of the paper, we study the correlation complexity of some common planning domains. Towards this end, we first establish some general properties of potential heuristics, concluding in two criteria to show that a task has correlation complexity at least 2. We begin with a result that is related to the incremental computation of potential heuristics.

Theorem 3. *Let* φ *be a potential heuristic for a planning task* Π*. Let* s *be a state of* Π*, let* o *be an operator applicable in* s*, and let* $s' = s[\![o]\!]$*. Then:*

$$\varphi(s') - \varphi(s) = \sum_{\substack{F \in \mathcal{F} \\ \mathit{vars}(F) \cap \mathit{vars}(\mathit{eff}(o)) \neq \emptyset}} w(F)([F \subseteq s'] - [F \subseteq s])$$

Proof: All other features are either present in both s and s' or absent in both s and s'. Their weights cancel out in the difference. □

Consider a heuristic h and an operator o applicable in a state s. We say that o is *good in* s under h if $h(s[\![o]\!]) < h(s)$ and *bad in* s under h otherwise. We say that a planning task is in *normal form* if $\mathit{vars}(\mathit{eff}(o)) \subseteq \mathit{vars}(\mathit{pre}(o))$ for all operators o [cf. Pommerening and Helmert 2015]. It is easy to see that for tasks in normal form, whether or not an operator is good under a potential heuristic of dimension 1 does not depend on the state s: either o improves the heuristic value in all states where it is applicable, or it does so in no state. Hence, for potential heuristics of dimension 1 we can speak of good or bad operators without referring to a specific state.

We say that operator o is *critical* in planning task Π if there exists an alive state s such that every s-plan includes o. (In other words, o is an action landmark in some alive state.)

Theorem 4. *Let* φ *be a descending potential heuristic of dimension 1 for a planning task* Π *in normal form.*

If o *is critical in* Π*, then* o *is good under* φ*.*

Proof: Because o is critical, there exists an alive state s from which every s-plan includes o. Because φ is descending, there exists a sequence of operator applications that reach a goal state from s and decrease the heuristic value in every step. All operators applied in this sequence must be good, and one of them must be o. □

If o has an inverse operator o' (i.e., $s[\![o]\!][\![o']\!] = s$ for some state s), then o and o' cannot both be good: if going from s to $s[\![o]\!]$ decreases the heuristic value, then returning from $s[\![o]\!]$ to s by applying o' must increase it to the original value. Together with Theorem 4 we obtain the first criterion for showing that a task cannot have correlation complexity 1.

Theorem 5. *Let Π be a planning task in normal form, and let o and o' be critical operators of Π that are inverses of each other. Then Π has correlation complexity at least 2.*

Proof: Assume the contrary: there exists a descending potential heuristic φ of dimension 1. From the previous theorem, o and o' are both good under φ. Inverse operators cannot both be good: a contradiction. □

For the second criterion, we need the notion of *dangerous* operators. Operator o is dangerous in task Π if there exists an alive state s in which o is applicable and $s[\![o]\!]$ is unsolvable.

Theorem 6. *Let Π be a planning task in normal form, and let o be an operator that is critical and dangerous in Π. Then Π has correlation complexity at least 2.*

Proof: Assume the contrary: there exists a descending potential heuristic φ of dimension 1 that avoids dead ends. Since o is critical, it is good under φ (Theorem 4). But o is also dangerous and hence leads from an alive to an unsolvable state. By the definition of avoiding dead ends, this means that o cannot be good: a contradiction. □

Spanner

We now begin our case studies of planning domains. In the Spanner domain (IPC 2014), an agent has to walk to a gate along a chain of m locations l_1–l_2–...–l_m, with the gate at l_m. At the gate there are n nuts that the agent has to tighten with n single-use spanners that it must pick up along the way. The agent can only move towards the gate, not backwards.

Lemma 1. *Spanner has correlation complexity at least 2.*

Proof: Consider a task with two locations l_1, l_2 and one spanner at l_1. Walking from l_1 to l_2 is critical, but dangerous. (Walking before picking up the spanner leads to an unsolvable state.) The result follows with Theorem 6. □

Theorem 7. *Spanner has correlation complexity 2.*

Proof: Let Π be a Spanner task with n spanners and m locations. For any location l_i let S_i be the number of spanners at all locations l_j with $j < i$. Walking to location l_i while carrying fewer than S_i spanners leads to an unsolvable state.

The following weight function defines a descending, dead-end avoiding potential heuristic φ of dimension 2 for Π. The result then follows with the preceding lemma.

$$w(\{\langle agent, l_i\rangle\}) = \sum_{k=1}^{i}(S_k - 1)$$
$$w(\{\langle agent, l_i\rangle, \langle carry\text{-}spanner_j, yes\rangle\}) = m - i$$
$$w(\{\langle carry\text{-}spanner_j, yes\rangle\}) = -m$$
$$w(\{\langle tightened_j, yes\rangle\}) = -m - 1$$
for all $1 \leq i \leq m$ and $1 \leq j \leq n$

We show that φ is descending and avoids dead ends by showing that the heuristic difference induced by picking up a spanner or tightening a nut is always negative and the potential difference induced by walking from l_{i-1} to l_i is negative iff the agent is carrying S_i spanners.

Picking up spanner j at location l_i changes the potential by $m - i - m = -i < 0$.

Tightening nut j is always done in location l_m and changes the potential by $-m - 1 - (-m) - (m - m) = -1$.

Walking from l_{i-1} to l_i while carrying s spanners changes the potential by $\sum_{k=1}^{i}(S_k - 1) - \sum_{k=1}^{i-1}(S_k - 1) + s((m - i) - (m - (i-1))) = S_i - s - 1$ which is negative iff $s \geq S_i$. □

Gripper

In the Gripper domain (IPC 1998), a robot with two grippers has to move n balls from room A to room B. It can pick up and drop balls in either room and move between the two rooms. The robot always starts in room A and the goal is always to transport all balls to room B. In a SAS$^+$ representation there is a variable specifying the position of the robot $r \in \{A, B\}$ and variables for the position of each ball $b_i \in \{A, B, G_1, G_2\}$ for $1 \leq i \leq n$. G_1 and G_2 stand for the two grippers.

Lemma 2. *Gripper has correlation complexity at least 2.*

Proof: In a Gripper task with more than 2 balls, moving from A to B and moving from B to A are both critical operators, and they are inverses of each other. The result follows with Theorem 5. □

Theorem 8. *Gripper has correlation complexity 2.*

Proof: The following weight function defines a descending, dead-end avoiding potential heuristic of dimension 2 for the Gripper task with n balls. The result then follows with the preceding lemma.

$$w(\{\langle r, B\rangle\}) = 1$$
$$w(\{\langle b_i, A\rangle\}) = 8$$
$$w(\{\langle b_i, G_j\rangle\}) = 4$$
$$w(\{\langle r, B\rangle, \langle b_i, G_j\rangle\}) = -2$$
for $i \in \{1, \ldots, n\}$ and $j \in \{1, 2\}$

The heuristic avoids dead ends because Gripper is undirected and hence harmless. To show that the heuristic is descending, we show by case distinction that every reachable non-goal state has an improving successor.

If the robot is in room A and can pick up a ball, picking it up changes the potential by $-w(\{\langle b_i, A\rangle\}) +$

$w(\{\langle b_i, G_j\rangle\}) = -8 + 4 = -4$. If there is no ball to pick up, but the robot has $g > 0$ balls in its grippers, moving to room B changes the potential by $w(\{\langle r, B\rangle\}) - g \cdot w(\{\langle r, B\rangle, \langle b_i, G_j\rangle\}) = 1 - 2g < 0$. If there are no balls to pick up and no balls in the grippers, the state is a goal state.

If the robot is in room B and has a ball in one of its grippers, dropping a ball changes the potential by $-w(\{\langle b_i, G_j\rangle\}) - w(\{\langle r, B\rangle, \langle b_i, G_j\rangle\}) = -4 - (-2) = -2$. If it does not have a ball in its grippers, moving to room A changes the potential by $-w(\{\langle r, B\rangle\}) = -1$. □

We remark that steepest ascent hill climbing with the given potential heuristic produces an optimal plan because picking up a ball in room A (improvement by 4) and dropping a ball in room B (improvement by 2) are always preferred to moving to the other room (improvement by 1).

VisitAll

In VisitAll (IPC 2011) an agent has to visit all vertices of a graph. In a SAS$^+$ encoding of the tasks there is a Boolean variable for each vertex indicating whether the vertex has been visited and a variable storing the position of the agent.

VisitAll tasks are not in normal form but we can transform them to normal form by replacing each operator *walk-A-B* with two operators: one for the case where B is already visited and one to visit B for the first time. The transformed domain has the same states and successor state relation and hence has the same correlation complexity as the original one.

Lemma 3. *VisitAll has correlation complexity at least* 2.

Proof: Consider a task with a chain of four locations (l_1–l_2–l_3–l_4) and initial location l_2. Consider the following two alive states s and s': in both states, l_2 and l_3 are the locations that have already been visited. In s, the agent is at l_2. In s', it is at l_3. From s, we see that *walk-to-visited*-l_2-l_3 is critical; from s', we see that its inverse *walk-to-visited*-l_3-l_2 is critical. The result follows with Theorem 5. □

Theorem 9. *VisitAll has correlation complexity* 2.

Proof: Let Π be a task with n locations $l_1, \dots, l_n$ forming a connected graph. (If the graph is unconnected, the task is unsolvable.) Let $d(i, j)$ be the shortest path distance between l_i and l_j. The following weight function defines a descending, dead-end avoiding potential function of dimension 2 for Π:

$$w(\{\langle \textit{visited-}l_i, \textit{no}\rangle, \langle \textit{pos}, l_j\rangle\}) = d(i, j)2^i \quad \text{for all } i, j.$$

The result then follows with the preceding lemma.

The function avoids dead ends because VisitAll is harmless. To show that it is descending, we consider a non-goal state where the unvisited location with the highest index is l_m. Moving one step in the direction of l_m decreases the potential by at least $2^m - \sum_{1 \le i < m} 2^i = 2$. □

We remark that even though the construction uses exponential weights, it leads to a polynomial planning algorithm because singly exponential numbers require only linear space to represent (hence computing the heuristic is not expensive), and the length of the generated plan is at worst quadratic in the number of locations. (It never takes more than n steps to reach another previously unvisited location.)

Blocksworld

In Blocksworld [e.g., Slaney and Thiébaux 2001] there are stacks of n blocks that must be rearranged from an initial to a goal configuration. We assume the following standard SAS$^+$ encoding for the domain formulation without an explicit hand: for each block A there is a Boolean variable *clear-A* denoting whether another block can be stacked on top of A and a variable *pos-A* that specifies what is below A. The possible values of *pos-A* are one value B for each other block B and the special value T for being on the table. Operators move a clear block from one block onto another, from a block onto the table, or from the table onto a block.

Lemma 4. *Blocksworld has correlation complexity at least* 2.

Proof: Consider a task with initial state A–B–D–C (A is on top of the tower) and goal A–B–C–D. Moving A from B to the table and its inverse are critical. We apply Theorem 5. □

Theorem 10. *Blocksworld has correlation complexity* 2.

Proof: Let Π be a Blocksworld task with blocks $\mathcal{B}$ where s_G is a goal state. We call the position of any block A in s_G its target position G_A (which may be the table). If a block is in its target position, it is *correctly placed*, otherwise *misplaced*. For each tower in s_G, we number the blocks from top to bottom, i.e., the top block B of each tower has $\textit{level}(B) = 1$, the block directly below it has *level* 2, etc. We call a block B *controlled* by a block A if B is anywhere below A in a tower of s_G. We say a block is *done* in a state if it and all blocks that are below it in s_G are correctly placed.

Blocksworld is undirected, and hence avoiding dead ends is trivial. The following weight function defines a descending potential heuristic of dimension 2 for Π. The result then follows with the preceding lemma.

Atomic features for all blocks $A \in \mathcal{B}$:

$$w(\{\langle \textit{pos-A}, X\rangle\}) = 2 \text{ for all } X \in \mathcal{B} \setminus \{A\}, X \neq G_A$$

$$w(\{\langle \textit{pos-A}, T\rangle\}) = \begin{cases} -1 & \text{if } G_A = T \\ 1 & \text{otherwise} \end{cases}$$

Conjunctive features for all blocks $A, B \in \mathcal{B}$ where B is controlled by A and all $X \in \textit{dom}(\textit{pos-B})$ with $X \neq G_B$:

$$w(\{\langle \textit{pos-A}, G_A\rangle, \langle \textit{pos-B}, X\rangle\}) = 2^{\textit{level}(A)}$$

In words, the conjunctive features punish situations where block A is correctly placed while block B controlled by A is misplaced. We now show that every reachable non-goal state s has an improving successor s'.

If all not-done blocks are on the table in s, consider a not-done block A where G_A is done. Moving A onto G_A reduces the heuristic value by $w(\{\langle \textit{pos-A}, T\rangle\}) = 1$.

Otherwise, s has a tower of at least two blocks such that the top block A is not done. Let B denote the block below A. Consider state s' reached by moving A onto the table.

If A is misplaced in s, then the atomic features change the heuristic value by $w(\{\langle \textit{pos-A}, T\rangle\}) - w(\{\langle \textit{pos-A}, B\rangle\})$ in s', which is -1 or -3 and hence an improvement. The conjunctive features can only change if A is correctly placed in s', which implies $G_A = T$. Then A controls no other blocks,

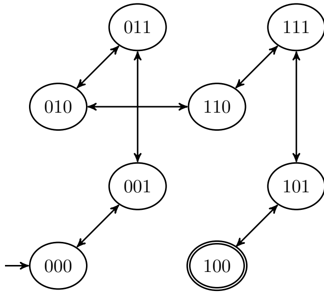


Figure 1: State space of a planning task with correlation complexity 3. The task has three binary variables v_1, v_2 and v_3, and a node with label xyz represents the state $\{\langle v_1, x\rangle, \langle v_2, y\rangle, \langle v_3, z\rangle\}$. Each edge represents an operator with three preconditions and one effect. The initial state is 000 and the only goal state is 100.

and hence no conjunctive feature becomes true. Conjunctive features related to blocks controlling A may become false, but this only improves the heuristic value further.

If A is correctly placed in s, the part of the heuristic value that is due to atomic features increases by 1 when going from s to s'. The change from conjunctive features is

$$\Delta = \sum_{\substack{\text{correctly placed } C \in \mathcal{B} \\ C \text{ controls } A}} 2^{\mathit{level}(C)} - \sum_{\substack{\text{misplaced } D \in \mathcal{B} \\ A \text{ controls } D}} 2^{\mathit{level}(A)}.$$

A controls at least one misplaced block D because A is not done in s, and hence the right sum is at least $2^{\mathit{level}(A)}$. The left sum is at most $\sum_{i=1}^{\mathit{level}(A)-1} 2^i = 2^{\mathit{level}(A)} - 2$, where the maximum is attained if all blocks controlling A are correctly placed in s. We get $\Delta \leq (2^{\mathit{level}(A)} - 2) - 2^{\mathit{level}(A)} = -2$. This compensates the increase of 1 from the atomic features: s' is an improving successor. This completes the proof. □

Similar to VisitAll, the potential heuristics give rise to a polynomial planning algorithm despite the exponential weights. It is easy to verify that hill-climbing with these potential heuristics moves each block at most two times (steepest ascent hill-climbing) or three times (simple hill-climbing).

Tasks with Higher Correlation Complexity

All the domains we studied so far have correlation complexity 2. The natural question is whether there are tasks with higher correlation complexity. We now answer this question in the affirmative by giving an example of a planning task with correlation complexity 3. The state space of the example task is shown in Figure 1. We obtained this task by mimicking the construction of the reflected binary code, also known as *Gray code* (Gray 1953). Gray code is based on nested layers of reflections, and because of these reflections, intuitively speaking, the state changes that need to be made in the example task in one half of the state space are exactly the opposite of the state changes that need to be made in the other half. This makes the "correct" operator to take heavily dependent on context and hence potential heuristics of low dimension cannot give sufficient guidance for this task.

Lemma 5. *The planning task in Figure 1 has correlation complexity at least 3.*

Proof: Any descending potential function for the task has to strictly decrease along the (unique) optimal plan. As the heuristic values are linear combinations of weights, each step in the plan yields a linear constraint over weights that is a necessary condition for a given potential function to be descending. For example, for the first step, we get the constraint

$$\begin{aligned} & w_{0\star\star} + w_{\star 0\star} + w_{\star\star 0} + w_{00\star} + w_{0\star 0} + w_{\star 00} \\ > & w_{0\star\star} + w_{\star 0\star} + w_{\star\star 1} + w_{00\star} + w_{0\star 1} + w_{\star 01}. \end{aligned}$$

Here, $w_{0\star\star}$ denotes the weight for the feature $\{\langle v_1, 0\rangle\}$, $w_{0\star 1}$ denotes the weight for the feature $\{\langle v_1, 0\rangle, \langle v_3, 1\rangle\}$, etc.

Using basic algebra or a solver for linear programs, we can verify that there is no solution that satisfies all constraints. □

Intuitively, the reason why potential heuristics of dimension 2 are not sufficient for the example is that one has to consider the values of both v_1 and v_2 to decide whether v_3 should be changed from 0 to 1 to advance towards the goal, or whether the opposite transition is needed. Moreover, this dependency on v_1 and v_2 cannot be expressed by linear combinations of v_1 and v_2 because the correct decision is governed by their exclusive-or combination, $v_1 \oplus v_2$.

Theorem 11. *The planning task in Figure 1 has correlation complexity 3.*

Proof: As mentioned in Section , the correlation complexity of a planning task is bounded from above by the number of state variables in the task. The result then follows with the preceding lemma. □

This result concludes our case studies. In the following sections, we compare correlation complexity to related concepts from the literature.

Relation to Persistent Hamming Width

Chen and Giménez (2007) introduced four related concepts for measuring the *width* of a planning task. Width is an indicator of complexity: they describe a planning algorithm that finds solutions for solvable planning tasks in time that scales exponentially (only) in the width of the task.

The most general of the width concepts considered by Chen and Giménez is *persistent Hamming width*. A planning task has persistent Hamming width k if it is unsolvable, or if from every reachable non-goal state s, it is possible to reach a state s' where the set of satisfied goals in s' is a strict superset of the set of satisfied goals in s, and none of the states on the path from s to s' differs from s in more than k state variables.

Unlike correlation complexity, which is defined for all planning tasks, persistent Hamming width is undefined for solvable planning tasks where an unsolvable state can be reached. The planning algorithm described by Chen and Giménez is incomplete when applied to such tasks. However, for planning domains with bounded width, it is a complete polynomial-time planning algorithm.

The work by Chen and Giménez resembles the state space topology study of Hoffmann (2005) in the sense that it measures how much work a search algorithm must perform to compensate for inaccuracies of a heuristic. (Even though Chen and Giménez do not explicitly consider heuristics, their search algorithm behaves similarly to enforced hill-climbing using a heuristic counting the number of unsatisfied goals.) In contrast, correlation complexity measures how complex a heuristic must be in order to guide a search algorithm directly to the goal. As the main purpose of the search component in a heuristic search algorithm is to compensate for inaccuracies of the heuristic, needing more complex heuristics vs. needing more search can be viewed as two faces of the same coin.

It is not hard to find examples where correlation complexity and persistent Hamming width widely disagree on the "difficulty" of a planning domain. This is to be expected: in both cases, the intuition is that low complexity means that solutions can be found efficiently (in the case of correlation complexity with the added difficulty that low complexity only means that accurate potential heuristics of low dimension exist, but does not tell us how to construct them). The converse is not necessarily true: if a domain has high persistent Hamming width (for example), this does not imply that planning is hard in this domain, only that the particular algorithm considered by Chen and Giménez might not be suitable for it.

A simple example of disagreement between the two measures are domains with reachable dead ends, like the Spanner domain (Section). It has correlation complexity 2, but no well-defined persistent Hamming width. On tasks with more than one spanner, the algorithm by Chen and Giménez will fail because it tries to achieve one of the goals as quickly as possible, which means picking up only one spanner and reaching a dead end.

The two measures can also disagree in domains without dead ends. As an example, consider a family of planning tasks where the n-th task encodes an n-ary binary counter counting backwards. We can encode this task with n state variables $\{v_{n-1}, \dots, v_0\}$, all with domain $\{0, 1\}$, set to 1 initially and required to be 0 in the goal. The state $\{\langle v_{n-1}, d_{n-1}\rangle, \dots, \langle v_0, d_0\rangle\}$ represents the counter value $\sum_{i=0}^{n-1} d_i 2^i$, and there are n operators that encode decrementing the counter by 1. (Each operator encodes one of the cases of $0, \dots, n-1$ carries.)

The correlation complexity for all these tasks is 1: using weight 2^i for the feature $\{\langle v_i, 1\rangle\}$ results in the perfect heuristic. The persistent Hamming width of the n-th task is n: from the state representing the counter value 2^{n-1}, all n state variables must be changed to make progress towards the goal.

In a later paper, Chen and Giménez (2009) generalized persistent Hamming width to *macro persistent Hamming width*, which additionally allows the use of macros computed on the fly that temporarily pass through states whose Hamming distance from the current state is larger than k. This modification leads to tractability results for some domains where no such results could be obtained for persistent Hamming width, such as a formulation of Blocksworld with an explicit arm. However, adding macros does not influence the overall greediness of the approach (trying to achieve each individual goal as quickly as possible), and hence the modified algorithm still gets trapped in dead ends in the Spanner domain. It also does not improve over persistent Hamming width in the binary counter domain, although it does lead to tractability in a formulation of Towers of Hanoi, where it generates (compact representations of) exponentially long plans in polynomial time (Chen and Giménez 2009).

Relation to Serialized Iterated Width

Lipovetzky and Geffner (2012; 2014) also introduced a notion of width for planning tasks. Very roughly speaking, according to their definition a planning task has *width* k if interactions between at most k facts must be considered in order to solve a planning task. Lipovetzky and Geffner observe that *optimal* solutions to a planning task can be found in time that is only exponential in the width of the task.

Most of the commonly considered planning domains do not admit polynomial-time optimal planning algorithms unless P $=$ NP (Helmert 2003), and consequently most planning domains do not have bounded width. To the best of our knowledge, no examples of planning domains with bounded width have been described in the literature. However, Lipovetzky and Geffner observe that many common benchmark domains have bounded width when restricted to the case where the goal is a single fact, and that many of them can be solved by *serialization*, focusing on one goal fact at a time. (This does not contradict the previously mentioned complexity result because such serialized solutions are not necessarily optimal, even if the plans for the individual goal facts are.) Based on this observation, they introduce the *Serialized Iterated Width* algorithm, which achieves polynomial runtime on a wide range of benchmark domains.

This notion of width and the Serialized Iterated Width algorithm do not give rise to polynomial algorithms in cases like the Spanner domain (Section) that require global resource reasoning. Spanner tasks with n spanners have width $\Theta(n)$ and cannot be serialized in the sense of Lipovetzky and Geffner, as focusing on one subgoal at a time and solving it optimally necessarily leads to a dead end. Similarly, the binary counter example from the previous section requires unbounded width to be solved with the Serialized Iterative Width algorithm: this is generally true for planning tasks where an exponential number of steps can be required to achieve the next goal fact. These are examples of domains with bounded correlation complexity but unbounded width.

However, it is also possible to construct tasks with low width and high correlation complexity. Given any planning task with correlation complexity n, we can create a new task (not equivalent to the original one) with width 1 by performing the usual conversion to a single goal fact (adding an artificial goal fact that can be achieved once the actual goal has been reached) and then adding a "cheating" operator that is only applicable in the initial state and directly achieves the artificial goal. The resulting task can be solved by a plan consisting only of the cheating operator and has width 1.

However, it still has correlation complexity n because correlation complexity considers *all* alive states, and hence having one obvious short solution does not automatically lead to low correlation complexity.

Conclusion

We introduced a new measure for the complexity of classical planning tasks. *Correlation complexity* measures how complex the features of a potential heuristic must be for the induced state space to contain no local minima.

Correlation complexity is a way to quantify how interrelated the state variables of a task are. Planning tasks for which it is necessary to take into account large conjunctions of facts have high correlation complexity. The benchmark planning domains we studied in this paper all have a low correlation complexity of 2. Given that potential heuristics with low dimension can be evaluated very efficiently, our results motivate further research on how to find good features and weights for potential heuristics automatically.

We also described an artificial planning task with correlation complexity 3, but so far we have no examples of "naturally occurring" planning domains that are tractable, yet have high correlation complexity. We believe that studying correlation complexity in a wider set of benchmark domains could be useful to further improve our understanding of what makes planning hard and what makes easy planning tasks easy.

Acknowledgments

This work was supported by the Swiss National Science Foundation (SNSF) as part of the project "Reasoning about Plans and Heuristics for Planning and Combinatorial Search" (RAPAHPACS).

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Computational Intelligence* 11(4):625–655.

Bonet, B., and van den Briel, M. 2014. Flow-based heuristics for optimal planning: Landmarks and merges. In *Proc. ICAPS 2014*, 47–55.

Brafman, R., and Domshlak, C. 2013. On the complexity of planning for agent teams and its implications for single agent planning. *AIJ* 198:52–71.

Chen, H., and Giménez, O. 2007. Act local, think global: Width notions for tractable planning. In *Proc. ICAPS 2007*, 73–80.

Chen, H., and Giménez, O. 2009. On-the-fly macros. In *Logic, Language, Information and Computation*, volume 5514 of *LNCS*, 155–169. Springer-Verlag.

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. ECP 2001*, 84–90.

Gray, F. 1953. Pulse code communication. US Patent 2,632,058.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. AIPS 2000*, 140–149.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proc. AAAI 2007*, 1007–1012.

Haslum, P.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *Proc. AAAI 2005*, 1163–1168.

Helmert, M., and Mattmüller, R. 2008. Accuracy of admissible heuristic functions in selected planning domains. In *Proc. AAAI 2008*, 938–943.

Helmert, M. 2003. Complexity results for standard benchmark domains in planning. *AIJ* 143(2):219–262.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Hoffmann, J. 2005. Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *JAIR* 24:685–758.

Keyder, E.; Hoffmann, J.; and Haslum, P. 2014. Improving delete relaxation heuristics through explicitly represented conjunctions. *JAIR* 50:487–533.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In *Proc. ECAI 2010*, 335–340.

Knuth, D. E. 1992. Two notes on notation. *American Mathematical Monthly* 99(5):403–422.

Lipovetzky, N., and Geffner, H. 2012. Width and serialization of classical planning problems. In *Proc. ECAI 2012*, 540–545.

Lipovetzky, N., and Geffner, H. 2014. Width-based algorithms for classical planning: New results. In *Proc. ECAI 2014*, 1059–1060.

Pommerening, F., and Helmert, M. 2015. A normal form for classical planning tasks. In *Proc. ICAPS 2015*, 188–192.

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. 2015. From non-negative to general operator cost partitioning. In *Proc. AAAI 2015*, 3335–3341.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the most out of pattern databases for classical planning. In *Proc. IJCAI 2013*, 2357–2364.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proc. ICAPS 2009*, 273–280.

Rintanen, J. 2012. Planning as satisfiability: Heuristics. *AIJ* 193:45–86.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence — A Modern Approach*. Prentice Hall.

Seipp, J.; Pommerening, F.; and Helmert, M. 2015. New optimization functions for potential heuristics. In *Proc. ICAPS 2015*, 193–201.

Slaney, J., and Thiébaux, S. 2001. Blocks World revisited. *AIJ* 125(1–2):119–153.

Suda, M. 2014. Property directed reachability for automated planning. *JAIR* 50:265–319.

Torralba, Á. 2015. *Symbolic Search and Abstraction Heuristics for Cost-Optimal Planning*. Ph.D. Dissertation, Universidad Carlos III de Madrid.

Duality in STRIPS planning*

Martin Suda
Institute for Information Systems, Vienna University of Technology, Austria

Abstract

We describe a duality mapping between STRIPS planning tasks. By exchanging the initial and goal conditions, taking their respective complements, and swapping for every action its precondition and delete list, one obtains for every STRIPS task its dual version, which has a solution if and only if the original does. This is proved by showing that the described transformation essentially turns progression (forward search) into regression (backward search) and vice versa.

The duality sheds new light on STRIPS planning by allowing a transfer of ideas from one search approach to the other. It can be used to construct new algorithms from old ones, or (equivalently) to obtain new benchmarks from existing ones. Experiments show that the dual versions of IPC benchmarks are in general quite difficult for modern planners. This may be seen as a new challenge. On the other hand, the cases where the dual versions are easier to solve demonstrate that the duality can also be made useful in practice.

1 Introduction

Propositional STRIPS language is one of the favourite formalisms for describing planning tasks. A STRIPS task description consists of an initial and goal condition formed by conjunctions of propositional atoms and of a set of actions made up by a precondition, add and delete lists. Despite its simplicity, the modelling power of the STRIPS formalism already captures the complexity class PSPACE (Bylander, 1994). Also, STRIPS lies in the core of the more expressive PDDL language (McDermott, 2000) used for representing benchmarks in the International Planning Competition.

Classical search is one of the basic but also most successful approaches to determining whether a given planning task has a solution. The search may proceed either in the forward direction starting from the initial state and applying actions until a goal state is reached, or in the backward direction where the goal condition is regressed over actions to produce sub-goals until a sub-goal satisfied by the initial state is obtained. Forward search is typically termed progression, while backward search is called regression.

*This research has been conducted at Max-Planck-Institut für Informatik, Saarbrücken, Germany. The author was also supported by the ERC Starting Grant 2014 SYMCAR 639270 and the Austrian research project FWF RiSE S11409-N23.

In this paper, we show that from the computational perspective there is no real difference between progression and regression in STRIPS planning. This is very surprising because progression is working with single states only while the sub-goal conditions in regression represent whole state sets. We show this result by describing a duality mapping working on the domain of all STRIPS planning tasks. Performing regression on the original task is shown equivalent to performing progression on the dual.

The existence of the duality mapping has some additional interesting consequences. For instance, any notion originally conceived and developed with one of the search approaches in mind has a dual counterpart within the other approach. We give examples of this phenomenon in Section 5, one of them being the dual of the relevance condition, an important ingredient in pruning the regression search space. The duality can also be used to construct new algorithms from old ones and to obtain new benchmarks from existing ones. Thus a purely theoretical concept at first sight, the duality also has immediate implications for practice.

The rest of the paper is organized as follows. After giving the necessary preliminaries in Section 2, we recall the details about progression and regression relevant for our work in Section 3. The duality mapping is defined and its properties are stated and proven in Section 4. We subsequently discuss immediate theoretical implications of the duality in Section 5. Section 6 then reports on our experiments. We compare the performance of several modern planners on dual versions of IPC benchmarks and also show how a planner can be adapted with the help of the duality to solve benchmarks previously out of reach. Finally, in the concluding Section 7, we discuss the applications of the duality from a broader perspective.

Previous work. The idea of inverting the search direction in planning was already considered by Massey (1999) in his dissertation. Our main theorem can be recovered from that work, where it follows from a more general, but perhaps a less elegant result. A proof similar to the one presented here can be found in Pettersson (2005).

Problem reversal was used by Haslum (2008) to enable progression-like reachability heuristics being used for regression search. Alcázar and Torralba (2015) use the same technique to compute backward invariants of planning prob-

lems. This is done, however, within the SAS$^+$ formalism and is therefore not directly comparable to our results.

It seems that although already known, the idea of duality is not very well known among the planning community. We hope that the discussion on both its theoretical and practical implications as well as the experimental evaluation presented in this paper will trigger further research on this interesting notion.

2 Preliminaries

A propositional STRIPS *planning task* is defined as a tuple $\mathcal{P} = (X, I, G, \mathcal{A})$, where X is a finite set of *atoms*, $I \subseteq X$ is the *initial condition*, $G \subseteq X$ is the *goal condition*, and $\mathcal{A}$ a finite set of *actions*. Every action $a \in \mathcal{A}$ is a triple $a = (pre_a, add_a, del_a)$ of subsets of X referred to as the action's *precondition*, *add list*, and *delete list*, respectively.

The semantics is given by associating each planning task $\mathcal{P} = (X, I, G, \mathcal{A})$ with a *transition system* $\mathcal{T}_\mathcal{P} = (S, I, S_G, T)$, where the set of *world states* $S = 2^X$ is identified with the set of all subsets of X, the *initial state* is the subset I, the *goal states* $S_G = \{s \in S \mid G \subseteq s\}$ are those states that satisfy the goal condition G, and, finally, the *transition relation* T, which consists of state-action-state triples called *transitions*, is defined as follows:

$$T = \{s \xrightarrow{a} s' \mid pre_a \subseteq s \wedge s' = (s \cup add_a) \setminus del_a\}.$$

A planning task *has a solution* if there is a *path* in the respective transition system from the initial state to a goal state, i.e. if there is a finite sequence of transitions $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \dots s_{k-1} \xrightarrow{a_k} s_k$ such that $s_0 = I$ and $s_k \in S_G$. Notice that the path π is fully determined by the sequence of actions $a_1, \dots, a_k$, which we call a *plan* for $\mathcal{P}$.

3 Progression and regression

There are two basic approaches to searching for solutions of planning tasks: progression and regression Russell and Norvig (2010). Progression, or simply forward search, proceeds systematically from the initial state and applies actions until a goal state is reached. Regression, or backward search, on the other hand, regresses the goal condition over actions to produce sub-goals until a sub-goal contained in the initial state is obtained.

In what follows we abstract away the actual search algorithm and only focus on properties of the two approaches that are important for showing their correctness. These properties depend solely on three "entry point" procedures, by which the actual search algorithm could be parameterized:

- $\texttt{start}()$, which generates a start *search node*,
- $\texttt{is_target}(t)$, which tests whether a given search node is a *target* node, and
- $\texttt{succ}(t)$, which generates *successor* nodes t' of the given search node t.

Given a plannig task $\mathcal{P} = (X, I, G, \mathcal{A})$, the respective implementations of the procedures for progression and regression are summarized in Table 1. Let us first focus on progression. There, each search node directly corresponds to a world state, or, more specifically, to a world state reachable from the initial state. The start search node $\texttt{start}^{Pr}()$ is equal to the initial state I itself, the $\texttt{is_target}^{Pr}(t)$ procedure tests whether the given node satisfies the goal condition, and the successor nodes $\texttt{succ}^{Pr}(t)$ are constructed by taking for every action $a \in \mathcal{A}$ for which the *applicability* condition $pre_a \subseteq t$ is satisfied the successor node $t' = (t \cup add_a) \setminus del_a$. This naturally corresponds to the definition of the transition system $\mathcal{T}_\mathcal{P}$ and so the proof of the following correctness theorem for progression becomes immediate.

Theorem 1. *A planning task $\mathcal{P} = (X, I, G, \mathcal{A})$ has a solution if and only if there exists a sequence of search nodes $t_0, \dots, t_k$ such that $t_0 = start^{Pr}()$, $is_target^{Pr}(t_k)$, and for every $i = 1, \dots, k$ $t_i \in succ^{Pr}(t_{i-1})$.*

In the case of regression, a search node is also represented by a subset of X, but it should be viewed as a sub-goal to be met, corresponding to a set of world states that satisfy it. Here, the search nodes are manipulated in the following way. The start search node $\texttt{start}^{Re}()$ is identified with the (sub-)goal G itself, the $\texttt{is_target}^{Re}(t)$ procedure returns true if and only if the initial state I satisfies t, and the successor search nodes $\texttt{succ}^{Re}(t)$ are generated by collecting the regressed sub-goals $t' = (t \setminus add_a) \cup pre_a$ for every action $a \in \mathcal{A}$ for which the *consistency* condition $del_a \cap t = \emptyset$ holds. The key property of regression is that in every world state s satisfying the regressed sub-goal t' (i.e., in every s such that $t' \subseteq s$) the action a is applicable ($pre_a \subseteq s$) and leads to a world state that satisfies the original sub-goal t. Consistency is needed to ensure that the action does not undo any desired atom.

Remark. Another property that is typically required, apart from consistency, is relevance. An action $a \in \mathcal{A}$ is said to be *relevant* for achieving a sub-goal t if and only if $add_a \cap t \neq \emptyset$, i.e., if when applied, it achieves a part of the sub-goal. Because relevance is only important for efficiency and not for correctness of algorithms based on regression, we set it aside for now, to keep things simple, and return to it in a later discussion.

The correctness theorem for regression has exactly the same form as the one for progression. We do not detail its proof, which is standard and basically just combines the insights mentioned above.

Theorem 2. *A planning task $\mathcal{P} = (X, I, G, \mathcal{A})$ has a solution if and only if there exists a sequence of search nodes $t_0, \dots, t_k$ such that $t_0 = start^{Re}()$, $is_target^{Re}(t_k)$, and for every $i = 1, \dots, k$ $t_i \in succ^{Re}(t_{i-1})$.*

4 Duality

When looking at Table 1, which compares progression and regression, it is not difficult to observe certain formal similarities. For instance, the role played by the initial condition I in progression is similar to the one played by G in regression and vice versa. Similarly, the precondition pre_a and the delete list del_a of the considered action a seem to be exchanging roles in a certain way. In this section we describe an involutory mapping $_^d : STRIPS \to STRIPS$ acting on

	progression: $_^{Pr}$	regression: $_^{Re}$
start()	I	G
is-target(t)	$G \subseteq t$	$t \subseteq I$
succ(t)	$\{ t' \mid \exists a \in \mathcal{A} \,.\, pre_a \subseteq t \wedge t' = (t \cup add_a) \setminus del_a \}$	$\{ t' \mid \exists a \in \mathcal{A} \,.\, del_a \cap t = \emptyset \wedge t' = (t \setminus add_a) \cup pre_a \}$

Table 1: Instantiating progression and regression for a plannig task $\mathcal{P} = (X, I, G, \mathcal{A})$.

the class of all STRIPS planning tasks that shows that the above similarities are not a coincidence and that progression and regression are more closely related than is would seem at first sight.

For an action $a = (pre_a, add_a, del_a)$ a dual action a^d is formed by exchanging the precondition and delete list: $a^d = (del_a, add_a, pre_a)$. For a set of actions $\mathcal{A}$ the set of dual actions is $\mathcal{A}^d = \{a^d \mid a \in \mathcal{A}\}$. Now, given a planning task $\mathcal{P} = (X, I, G, \mathcal{A})$ the dual task $\mathcal{P}^d$ is obtained by exchanging the initial and goal conditions while taking their complements with respect to X, and using the dual action set:

$$\mathcal{P}^d = (X, (X \setminus G), (X \setminus I), \mathcal{A}^d).$$

It is easy to see that a mapping $_^d$ defined in this way is indeed *involutory* on the set of STRIPS planning tasks, meaning that $(\mathcal{P}^d)^d = \mathcal{P}$ for every task $\mathcal{P}$. This justifies the use of the term duality.

We can now state the central theorem about duality.

Theorem 3. *For every planning task $\mathcal{P} = (X, I, G, \mathcal{A})$ the dual task $\mathcal{P}^d$ has a solution if and only if $\mathcal{P}$ does. More specifically, a sequence of actions $a_1, \ldots, a_k$ is a plan for $\mathcal{P}$ if and only if the sequence $a_k^d, \ldots, a_1^d$ is a plan for $\mathcal{P}^d$.*

Proof. If a planning task has a solution, it can be found by both progression and regression, because they are both correct (Theorem 1 and 2). We prove this Theorem 3 by showing that regression for $\mathcal{P}$ performs exactly the same operations as progression for $\mathcal{P}^d$ when the search nodes are represented in a complemented form for the latter, i.e. when storing $X \setminus t$ in place of t. This is done in three steps corresponding to the three "entry point" procedures of Table 1.

First, we realize that

$$\mathtt{start}^{Re}_{\mathcal{P}}() = X \setminus \mathtt{start}^{Pr}_{\mathcal{P}^d}().$$

In words, the start search node of regression for $\mathcal{P}$, is the complement (with respect to X) of the start search node of progression for $\mathcal{P}^d$. Similarly, a search node $t \subseteq X$ is a target node in regression for $\mathcal{P}$ if and only if $(X \setminus t)$ is a target node in progression for $\mathcal{P}^d$:

$$\mathtt{is_target}^{Re}_{\mathcal{P}}(t) = \mathtt{is_target}^{Pr}_{\mathcal{P}^d}(X \setminus t),$$

which follows from the equivalence

$$a \subseteq b \leftrightarrow (X \setminus b) \subseteq (X \setminus a).$$

Finally, the successor nodes of a search node $t \subseteq X$ in regression for $\mathcal{P}$ can be computed as complements of successor nodes of $(X \setminus t)$ in progression for $\mathcal{P}^d$:

$$\mathtt{succ}^{Re}_{\mathcal{P}}(t) = \{(X \setminus t_0) \mid t_0 \in \mathtt{succ}^{Pr}_{\mathcal{P}^d}(X \setminus t)\}.$$

For this last point, it is sufficient to verify for every action $a \in \mathcal{A}$ that 1) the consistency condition in regression for $\mathcal{P}$ and applicability condition in progression for $\mathcal{P}^d$ are each other's dual:

$$\begin{aligned} del_a \cap t = \emptyset \ &\leftrightarrow \ del_a \subseteq (X \setminus t) \\ &\leftrightarrow \ pre_{a^d} \subseteq (X \setminus t), \end{aligned}$$

and, 2) regressing t over a yields the complement of applying a^d to the complement of t:

$$\begin{aligned} X \setminus ((t \setminus add_a) \cup pre_a) &= ((X \setminus t) \cup add_a) \setminus pre_a \\ &= ((X \setminus t) \cup add_{a^d}) \setminus del_{a^d}. \end{aligned}$$

With these two properties checked (by applying De Morgan's laws for sets) the theorem is proven by induction over k, the length of a solution path $\pi = s_0 \xrightarrow{a_1} s_1 \ldots s_{k-1} \xrightarrow{a_k} s_k$ and the corresponding plan $a_1, \ldots, a_k$. □

5 Implications

The most striking consequence of Theorem 3 is the discovery that in STRIPS planning there is no substantial difference between progression and regression. Indeed, any algorithm based on one of the two approaches may be effectively turned into an algorithm based on the other by simply applying the duality mapping to the input as a preprocessing and running the actual algorithm on $\mathcal{P}^d$ instead of on $\mathcal{P}$. This transformation obviously preserves the length of the shortest plan and its cost.

Given this perspective, it is now interesting to observe what are the dual counterparts of notions that were originally conceived and developed with only one of the approaches in mind and in how do they emerge "on the other side of the duality". We will now comment on some of these observations in the following subsections.

Relevance and usefulness

It was mentioned before that it is important for the efficiency of regression to only regress over actions that are relevant for the current sub-goal. Let us repeat that an action $a \in \mathcal{A}$ is relevant for t if and only if $add_a \cap t \neq \emptyset$. Regressing over an action that is not relevant for t results in a (possibly strictly) stronger sub-goal $t' \supseteq t$. We may safely discard t' from consideration, because successfully regressing t' is (possibly strictly) more difficult than successfully regressing t.[1] This way filtering out non-relevant actions helps to keep the regression search space manageable.

It is now at hand to ask what the dual notion of relevance is. For lack of a better word, we call it usefulness. We say

[1] If solution can be found from t', it can be found from t as well.

that an action $a \in \mathcal{A}$ is *useful* in a state t if and only if the add list of a is not fully contained in t. We see that usefulness is a natural property: it does not make sense to progress via a non-useful action, because it will never make more atoms true in the resulting state. The reason why usefulness is generally not mentioned in the literature is that in typical benchmarks there are seldom actions that would be applicable and yet not useful in a given state. This is in contrast with regression where consistency and non-relevance are far less correlated.

First add, then delete?

When defining the result of action application to a state, one needs to decide in which order should the add list and the delete list be considered. In particular, if a description of a planning task contains an action a such that add_a and del_a have a non-empty intersection, the result of applying a to a state s depends on this order. One can either exclude this possibility up front by requiring that for any action the add and delete lists are disjoint, or, alternatively, to decide on a canonical order of their application.

There are two remarks we can make here with respect to duality. First, if we choose the former option above, i.e., if we require that $add_a \cap del_a = \emptyset$ for any $a \in \mathcal{A}$, we should perhaps (for the sake of symmetry) also require that $add_a \cap pre_a = \emptyset$, because that is exactly the condition under which the order of applying add list and the precondition during regression of a sub-goal becomes irrelevant. Note that this condition also makes sense from the perspective of progression, because atoms mentioned in the precondition will be preserved by the action anyway (unless deleted) so they do not need to be mentioned again in the add list.

The second remark relates to the latter option, when in order to resolve the above situation a particular add-delete order is chosen as canonical. Here the duality dictates (with appeal to elegance of the theory) that adding should happen before deleting, as done in our definition in Section 2. It is because only with that order the proof of Theorem 3 goes through as presented.

Let us be more specific. In progression we, quite naturally, first check the applicability condition $pre_a \subseteq s$, before applying the effects. That is why the corresponding regression operation needs to first subtract the add list from the sub-goal, before adding the preconditions: $t' = (t \setminus add_a) \cup pre_a$. Then dualizing the last equation gives us $s' = (s \cup add_a) \setminus del_a$ as promised. This should not be interpreted as saying that the duality itself relies on a particular ordering of addition and deletion in the definition of action application. Should the other order be adopted instead, however, we would need to require that the actions of a planning task are normalized beforehand so that the intersection of add and delete lists is always empty.

Semantics of search nodes

Since the duality exchanges the roles of progression and regression, one should ask what happens to the semantics of the search nodes, which are known to represent world states in progression and sets of world states (via conjunctive conditions) in regression. The surprising answer the duality gives is that both the views are equally valid for both progression and regression. One just needs to go over to the complement representation to see the other.

Essentially, nodes in progression can be interpreted as a conditions, where a condition t stands for all the states s such that $s \subseteq t$, i.e. states having at most those positive facts as those stated in t, but no others. This is because in STRIPS, we can only make a task of reaching a goal harder by removing a fact from from a state.

Dually, regression can be thought of as performed over single states only, the states corresponding to the search nodes themselves, because we can only make regression harder by adding facts to such states. We invite the reader to check the details for herself by replaying the proof of Theorem 3 from this perspective.

Note that this observation provides us with a new way (arguably less intuitive, but nevertheless a legitimate one) to justify the correctness of the two approaches. While this may sometimes simplify argumentations, the actual implementation "mechanics" remains intact.

Limitations

We close this section by discussing the limitations of the duality concept. A careful analysis of the proof of Theorem 3 reveals that it substantially relies on the particularly simple form of regression in STRIPS planning. Essential is the fact that regressed sub-goals may be represented as conjunctions of atoms. This means the duality does not directly carry over to more expressive formalisms which allow negated goals or preconditions. For similar reasons, extending the duality to Finite Domain Representation (FRD) Helmert (2009) seems problematic. The good news is that the duality applies to the lifted version of STRIPS as realized by the STRIPS subset of the PDDL language McDermott (2000) used in the International Planning Competition (IPC).[2] The IPC benchmark set contains more than a thousand practically relevant problems to which the duality applies.

6 Experiments

The duality mapping we have described in the previous section provides us with a means of transforming one planning task into another while preserving the existence of its solution. It is now natural to ask how difficult are the dual versions of IPC benchmarks for modern planners. We performed a series of experiments in order to answer this question and we report on them in this section.

Note that there are two possible ways of interpreting the results. We may either view the dual versions as new standalone problems, or imagine the duality mapping as part of the algorithm we are currently testing. The second case may be understood as an evaluation of a new, dual algorithm on the original benchmarks. We will prefer the first view for most of this section, but adopt the second where it is more natural.

For our experiments, we collected all the benchmarks from the satisficing tracks of the International Planning

[2] To complement the initial and goal condition, one first obtains the set of all atoms X by grounding the domain predicates.

	FF	LAMA	Mp
ORIG	1009	1192	1114
DUAL	136	175	329

Table 2: First experiment: number of ORIG and DUAL problems solved within 180 seconds by the respective planners.

	FF	FF-U	FF-UI	FF-UIN
DUAL	136	204	682	695

Table 3: Second experiment: number of problems from DUAL solved within 180 seconds by modifications of the planner FF.

Competitions[3] (IPC) of years 1998–2011 that are in the STRIPS subset of the PDDL language.[4] Together we collected 1564 problems. We then used the preprocessing part of the planner FF Hoffmann and Nebel (2001) to produce a grounded version of these. Note that FF's relevance analysis was involved in the process, so all the "rigid" predicates that are only used for modelling purposes and the value of which is not affected by any action were removed. Let us denote the set of these grounded IPC benchmarks ORIG.

The preprocessing tool was then extended further to implement the duality mapping: It first normalizes the actions so that the precondition and delete list never intersect with the add list. To conform with the official IPC semantics, which is "first delete, then add" Fox and Long (2003), this is done by performing for every action a the following two assignments in the prescribed order:

$$del_a := del_a \setminus add_a; \quad add_a := add_a \setminus pre_a.$$

Then the duality mapping is applied. Let the problems obtained this way be denoted as DUAL. All the experiments were performed our servers with 3.16 GHz Xeon CPU, 16 GB RAM, with Debian 6.0.

In the first experiment we ran the following three planners on both ORIG and DUAL benchmark sets:

- the FF planner Hoffmann and Nebel (2001) as a baseline representative of heuristic search Bonet and Geffner (2001) planners,
- the Fast Downward planner Helmert (2006) in the configuration LAMA-2011 Richter and Westphal (2010), another heuristic search planner, the winner of the satisfycing track of the last IPC held in 2011, and
- the planner Mp Rintanen (2010), as a representative of the planning as satisfiability Kautz and Selman (1996) approach.

The time limit was set to 180 seconds per problem.

The results of the first experiment are summarized in Table 2. We see that the problems in DUAL are generally much more difficult to solve than ORIG, and that the SAT-based planner Mp performs better on DUAL than the heuristic search planners.

We conjecture (and later partially verify) the following reasons for the difficulty of DUAL. First, the explicit state forward search planners suffer from not testing for usefulness of actions. This corresponds to omitting the relevance test in the dual, regression-based algorithm and makes the search space unnecessarily large. The second reason is that *invariant* information is no longer recovered from the task description by the planners. Invariant is a property which holds in the initial state and is preserved by all transitions. While logically redundant, invariants are known to be usually critical for efficiency of SAT-based planners Rintanen (2010). Moreover, the existence of simple invariants formed by negative binary clauses is a prerequisite for the reconstruction of a non-trivial Finite Domain Representation (FDR), which LAMA is attempting to build in its preprocessing phase Helmert (2009). As we independently checked, there are almost no binary clause invariants to be recovered from the DUAL benchmarks. This means that Mp has to search for plans without the useful guidance the invariants usually provide and LAMA most of the time discovers only trivial, two-valued domains for its finite domain variables.

Remark. Note that the problems in DUAL still contain the original invariant information, but it has been turned into *backward invariants*, properties of the goal states preserved when traversing the transitions backwards. Obviously, the planners do not check for backward invariants, because typically, e.g., on ORIG, it does not pay off.

In our second experiment, we set out to discover to what extent do the above reasons explain the degraded performance of the planners on DUAL. We focused on the planner FF for its relative simplicity and modified it in several steps in order to make it perform better on DUAL. We prepared the following versions of the planner:

- FF-U, which checks for usefulness of actions and discards the non-useful ones,
- FF-UI, which additionally computes[5] binary clause backward invariant, and discards successor states that violate it,
- FF-UIN, which additionally turns off enforced hill climbing (see Hoffmann and Nebel (2001)) and always directly starts best first search.[6]

We ran all the modifications on DUAL, again with the time limit of 180 seconds per problem.

The numbers of problems solved by the respective modifications are shown in Table 3. For the sake of comparison we also repeat the result for the original FF. It can be seen that each of the modifications represents an improvement over the previous version. Probably the most is gained by incorporating the backward invariant test. Actually, each modification solves a strict superset of the problems solved

[3] http://ipc.icaps-conference.org/

[4] We dropped the action cost feature where present.

[5] We use an efficient implementation of the fixpoint algorithm described in Rintanen (1998).

[6] We observed that enforced hill climbing fails on most of the problems in DUAL, so turning it off up front saves some time.

		FF (unique)	FF-DUAL (unique)
PSR	(50)	39 (2)	45 (8)
Woodworking	(50)	18 (2)	44 (28)
Floortile	(20)	7 (0)	17 (10)

Table 4: Third experiment: Comparing FF and FF-DUAL on three domains where the latter dominates the former. Size of each domain and the number of problems uniquely solved by the respective planner are shown in parenthesis.

by the previous one. An exception is the last step where FF-UI solves 3 problems that FF-UIN cannot solve. However, FF-UIN solves 16 problems that FF-UI cannot solve within the given time limit.

Despite our efforts to improve the performance of FF on `DUAL`, the planner still solves less problems from `DUAL` than from `ORIG`. In our third experiment, we tried to discover whether there are some problems in `DUAL` that the improved FF-UIN can solve, while the original FF fails on their counterparts in `ORIG`. This corresponds to the question whether the duality can be made useful in practice by helping to solve difficult IPC benchmarks. To simplify the following discussion, let us call by FF-DUAL a planner composed by the preprocessor, which grounds and dualizes inputs, followed by FF-UIN. We will now compare FF and FF-DUAL on `ORIG`.

Apart from six problems from the Mystery domain, where FF-DUAL correctly discovers that no plan can exists while FF timeouts, there are three domains where FF-DUAL performs consistently better than FF. Table 4 reports on the number of problems solved, categorized by the domains.

In order to better understand the success of FF-DUAL on the three domains, we more closely analyzed and compared the output of the two versions of the planner. In particular, we focused on the reported heuristic value of the currently expanded state. We noticed the following facts.

- On the domain PSR the heuristic value of the initial state is quite low (between 1 and 10). This holds for both FF and FF-DUAL, but the value for FF-DUAL is typically one higher than that for FF. In other words, the dual version of relaxed plan heuristic is more informative on PSR.
- On Woodworking, the heuristic value of the initial state ranges from 5 up to about 70. FF-DUAL's values are typically not higher, but stay quite close to those of FF.
- Although on Floortile, FF's heuristic is more informed than FF-DUAL's, FF's goal agenda mechanism seems to be making suboptimal decisions in decomposing the goal into sub-goals. On three problems where FF's enforced hill climbing fails within the time limit and the goal agenda is discarded, FF then successfully finds a plan with best first search. At the same time, FF-DUAL directly looks for a plan using best first search and its less informed heuristic.

On all the other domains FF-DUAL's heuristic value of the initial state is typically much lower than the corresponding estimate of FF. This might explain the general lower effectiveness of FF-DUAL on the `ORIG` benchmarks.

To sum up, in our experiments we have shown that the dual versions of IPC benchmarks are in general much more difficult to solve by modern planners than the originals. This can be partially remedied by adapting a planner to make use of specific features the dual benchmarks possess, but which are usually missing in the standard ones. Although the imagined dualizing planner FF-DUAL does not beat the original FF in the overall number of solved problems, there are certain domains where it indeed pays off to apply the duality mapping before looking for a plan. This represents one possible application of the duality concept in practice.

7 Conclusion

In this paper, we have described a duality mapping on the domain of all STRIPS planning tasks. Its existence shows that computationally, there is no real difference between performing progression and regression as they are each other's dual. Differences between the two that one can measure in practice follow from asymmetries (with respect to the mapping) of the concrete benchmarks and are not inherent to the search paradigms themselves. We believe that understanding these asymmetries and their influence on the efficiency of planning algorithms deserves further study.

Furthermore, we have pointed to several applications of the duality itself. We have shown that new theoretical insights may be obtained by translating known notions via the mapping and analyzing the obtained duals. For instance, there necessarily exists a "precondition relaxation heuristic" a dual of the famous delete relaxation heuristic.

Next we studied the dual versions of the standard IPC benchmarks and discovered they are quite difficult to solve for modern planners. One could argue that there is nothing interesting about difficult benchmarks in themselves if they do not come from practical applications – for instance, random problems form the phase transition region (see Rintanen (2004)) seem to have this status. We, however, do not think the dual IPC benchmarks fall into the same category. After all, they still encode the same transition structures as the originals, albeit in a non-obvious way. Therefore, we believe they should be considered as an auxiliary test set by anyone attempting to develop a really versatile planner.

Finally, we explored the possibility of using the duality to design new algorithms. A simple modification of the planner FF which uses the duality was shown to improve over the original system on several benchmark domains. Note that this obvious schema of first dualizing the input and then running a known algorithm is not the only option of how the duality can be used. More sophisticated algorithms combining progression and regression tied together by the duality can be imagined.

References

Vidal Alcázar and Álvaro Torralba. A reminder about the importance of computing and exploiting invariants in planning. In *ICAPS 2015*, pages 2–6. AAAI Press, 2015.

Blai Bonet and Hector Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33, 2001.

Tom Bylander. The computational complexity of propositional STRIPS planning. *Artif. Intell.*, 69(1-2):165–204, 1994.

Maria Fox and Derek Long. Pddl2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

P. Haslum. Additive and reversed relaxed reachability heuristics revisited. In *6th International Planning Competition Booklet (ICAPS-08)*, 2008.

Malte Helmert. The Fast Downward planning system. *J. Artif. Intell. Res. (JAIR)*, 26:191–246, 2006.

Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5-6):503–535, 2009.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *J. Artif. Intell. Res. (JAIR)*, 14:253–302, 2001.

Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In William J. Clancey and Daniel S. Weld, editors, *AAAI/IAAI, Vol. 2*, pages 1194–1201. AAAI Press / The MIT Press, 1996.

Bart Massey. *Directions In Planning: Understanding The Flow Of Time In Planning*. PhD thesis, University of Oregon, 1999.

Drew V. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2):35–55, 2000.

Mats Petter Pettersson. Reversed planning graphs for relevance heuristics in AI planning. In *Planning, Scheduling and Constraint Satisfaction: From Theory to Practice*, volume 117 of *Frontiers in Artificial Intelligence and Applications*, pages 29–38. IOS Press, 2005.

Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res. (JAIR)*, 39:127–177, 2010.

Jussi Rintanen. A planning algorithm not based on directional search. In Anthony G. Cohn, Lenhart K. Schubert, and Stuart C. Shapiro, editors, *KR 2004*, pages 617–625. Morgan Kaufmann, 1998.

Jussi Rintanen. Phase transitions in classical planning: An experimental study. In Didier Dubois, Christopher A. Welty, and Mary-Anne Williams, editors, *KR 2004*, pages 710–719. AAAI Press, 2004.

Jussi Rintanen. Heuristics for planning with SAT. In David Cohen, editor, *CP 2010*, volume 6308 of *LNCS*, pages 414–428. Springer, 2010.

Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

Improving Performance by Reformulating PDDL into a Bagged Representation

Pat Riddle, Jordan Douglas, Mike Barley
Department of Computer Science
University of Auckland
Auckland, New Zealand

Santiago Franco
Dept. de Informatica
Universidade Federal de Vicosa
Vicosa, Brazil

Abstract

This paper describes Baggy - a system which automatically transforms a PDDL representation into a revised PDDL representation, solves the problem using the revised representation, and transforms the solution back into the original representation. The basic approach involves counting objects that are indistinguishable, rather than treating them as individual objects. This eliminates some unnecessary combinatorial explosion. We report encouraging results on a number of IPC11/14 domains and give algorithm details including soundness proof sketches. We conclude by discussing related work and outlining plans for future research.

Researchers have shown that changing the PDDL representation can result in very different behavior by the same planner (Riddle, Barley, and Franco 2015; Riddle, Holte, and Barley 2011). In this paper we describe Baggy - a system which reformulates the PDDL representation of a given problem to reduce the size of the state space. In the original PDDL representation, each object has a unique identifier (id). However, in many domains some objects are indistinguishable in the initial state. Consider a simple transport domain: if there are 5 packages at the same location in the initial state then it does not matter which package the planner decides to load into the truck. A plan to move a package can use any package interchangeably. In this case, referring to each of the 5 packages with a unique identifier is contributing to unnecessary combinatorial explosion. In the initial state of the original representation there are 5 applicable grounded operators to load one of these packages onto the truck. In the reformulated representation there is only 1 applicable grounded operator. This operator decreases the number of packages at the location by 1 and increases the number of packages in the truck by 1 - without referring to the object ids. Such a transformation cannot reduce a problem's inherent complexity, but it can reduce "accidental complexity" in Haslum's (2007) sense. Furthermore, the transformation is done on the lifted representation, so less effort may be needed in the grounding process which creates SAS$^+$ variables (Bäckström and Nebel 1995). The PDDL produced by Baggy can be used by any PDDL planner, thus the approach differs from other techniques for symmetry reduction, which require alterations to the planner itself[1].

In this paper, there are 3 main contributions. We present Baggy and show it is a sound method to transform PDDL, which always results in a smaller state space. We show that there are situations where a state-of-the-art heuristic problem solver can perform better in the reformulated representation than in the original representation. Lastly we show that this is not always the case; for some problem/planner combinations, the new representation can perform worse.

A System for Reformulating PDDL

Baggy operates in a three step process. Baggy (1) determines which objects can be bagged and then **reformulates** the PDDL representation accordingly. If bagging takes place then (2) the reformulated PDDL domain and problem files can be parsed and solved by any planner to give a plan. Then Baggy (3) **transforms** the plan from reformulated space back into original space with a simple and fast mapping algorithm. It is important for the reader to bear in mind that a single bagged state maps to more than one original space state. Therefore a single bagged plan maps to more than one original plan, only one of which is guaranteed to be a solution to the original problem. The combination of reformulating the PDDL and transforming the solution path is sound.

Definitions The following definitions are used by both Baggy and by the proof sketches.

PDDL problem space (PS): The 6-tuple: <Types, Predicates, Actions, Objects, Initial state, Goal>.

Indexable: Type t is indexable if no predicate can have more than one type t argument.

Non-Negated: Type t is non-negated if there are neither negated goals nor negated operator preconditions containing an argument of this type. An exception to this is $\neq$.

Monotonicity Invariant: Let $C = \langle \phi, V \rangle$ where $\phi = \{\phi_1, \phi_2, \ldots, \phi_n\}$ is a set of predicates and $V = \{V_1, V_2, \ldots, V_n\}$ is a set of variable sets such that each element in V_i is an argument of ϕ_i. Then C is a monotonicity invariant if the total number of instances of ϕ with respect to each respective

[1] The experimental results in this paper have one alteration of the planner. Baggy passes a list of invariants to "translate" so that it makes fewer variables. It will work without this addition but it will make more SAS+ variables in some of the reformulated domains.

variable in V is non-increasing over all states reachable from the initial state. This definition and the algorithm for finding monotonicity invariants have been borrowed from Helmert (Helmert 2008).

Single-valued: Type t is single-valued if for every predicate, p, which has an argument of type t, (1) there exists a *monotonicity invariant*, $C = \langle \phi, V \rangle$, such that for some i, $p = \phi_i$ and there exists a variable in V_i of type t; and (2) the number of instances of C in the initial state, with respect to each object of type t, is 0 or 1. For example, boxes are single-valued, which means that in any state a box cannot be the subject of more than one *in* predicate, e.g., there cannot be both *(in box1 rm1)* and *(in box1 rm2)*.

Action Equivalent: A type is action equivalent if no action mentions any object of that type by its object id. For example, *box* is not action equivalent if some action discriminates between *box1* and *box2*.

Baggable: Type t is baggable if it is *indexable*, *non-negated*, *single-valued*, and *action equivalent*.

Attribute: The set of *monotonicity invariants* of a *baggable* type. These attributes describe everything about an object of a *baggable* type in any state.

Initial State Equivalent (ISE): A set of objects are ISE if they have the same *attribute* values in the initial state.

Goal Equivalent (GE): A set of objects are GE if they have the same *attribute* values in the goal description.

State Equivalent: A set of objects are state equivalent if they are all *ISE* or all *GE*. Given a predicate, e.g., *(between box1 rock hardSpot)*, *between* is the attribute id, *box1* is the baggable object, and the attribute value is *<rock hardSpot>*. If the predicate only has has one argument, e.g., *(clear box1)* then the attribute value is a boolean value.

Bag: A set of pairwise *state equivalent* objects of the same *baggable* type.

Macropredicate: A predicate which describes everything about a *bag* in a state. The first argument is the bag id, followed by one argument for each *attribute* and then finally the count (i.e. the number of objects which are in this bag and have these *attributes*).

Overview In Symmetry Reduction they use automorphisms of the state space stabilized with respect to the initial state and the goal, or in more recent work just the goal state. ISE and GE relations guarantee that the symmetries are stabilized with respect to the initial state or the goal, respectively. Our technique goes one step further than symmetry reduction and replaces the objects, that are determined to be indistinguishable, with a count variable. This is equivalent to the reformulation done by Amarel in the Missionaries and Cannibals problem (Amarel 1971). This prevents us from having to test for symmetry during the search itself. This does not mean that our new representation always outperforms current symmetry reduction techniques, as is discussed in this paper's results.

Outline of Next 3 Sections The next 3 sections correspond to the 3 steps Baggy takes to solve a problem. The first part discusses the reformulation of a problem, what it means for that reformulation to be correct, and sketches an informal proof that the reformulation is correct. The second part discusses the planner itself. The third part discusses the transformation process and sketches an informal proof that if the reformulation is correct and the planner is sound then the transformation of the reformulated plan results in a solution to the originally formulated problem. These 3 sections constitute an informal proof of the soundness of this approach.

1) Problem Space Reformulation

Simplifications in our Description of the Algorithm The following simplifications are made for explaining Baggy. In practice, Baggy accomplishes these functions by compiling them into the PDDL domain/problem description. No modifications to the planner are necessary.

Combining Objects and Constants We will combine objects and constants and simply call them both "objects".

Closed World for Counts If there are no objects with a specific set of attributes, the count is 0.

Arithmetic Operations There is a predicate, (sum ?X ?Y ?Z), where ?X and ?Y must be instantiated with integers. If ?Z is uninstantiated then ?Z is instantiated to the sum of the values of ?X and ?Y and the predicate evaluates to true. If ?Z is instantiated and equals the sum of ?X and ?Y then the predicate evaluates to true else it evaluates to false.

Arithmetic Relations The following binary predicates exist (with customary semantics): $\neq, <, \leq, =, \geq, >$.

Using Variables in the Goal We will assume our planner can handle goals with variables. Variables may be named or anonymous, e.g., "$?bx$" or "_".

Simplifying Assumptions for Proof Sketch The following are assumptions made to simplify the proof. However, they are neither limitations of Baggy nor of the theory.

Reformulate One Type While Baggy can reformulate many types simultaneously, we simplify our discussion by only considering reformulating a single type, e.g., *box*.

Dealing with Initial State Equivalence Relation While Baggy can create bags using both GE and ISE relations, even within a type, we discuss bags based on the ISE relation.

One Bag Type Parameter Modified Per Operator We assume no operator modifies more than one baggable type parameter.

Flat Type Hierarchy While Baggy can reformulate types that are subtypes, we assume that the reformulated type neither is a subtype nor a supertype. For example, we do not make *box* a subtype of another type, e.g., a *container* type.

No No-ops While Baggy handles original problem space actions that remove and add the same attribute values for a reformulated object (i.e., not really changing the object because all its effects cancel each other out), we assume that the original action preconditions prohibit this.

Running Example 1 The example is a simplified domain.

```
Original Problem Space Definition:
<(:types box, room),
 (:predicates (in ?bx - box ?rm - room)),
 (:actions
    (mv_box
```

```
    (:params ?bx - box ?from ?to - room)
    (:precondition
      (and (not (= ?from ?to))
        (in ?bx ?from)))
    (:effect
      (and (not (in ?bx ?from))
         (in ?bx ?to))))),
 (:objects a b c d e - box r1 r2 r3 - room),
 (:init ((in a r1) (in b r1) (in c r1)
        (in d r1)(in e r3))),
 (:goal (and (in b r2) (in c r2)
            (in d r1) (in e r2)))>
```

Reformulated Problem Space Definition:

```
<(:types boxbag, room, integer),
 (:predicates  (bagState ?bx - boxbag ?rm - room ?cnt - integer)
                    (bag_size ?bx - boxbag ?size - integer)),
 (:actions
     (mv_box
     (:params ?bx - boxbag ?from ?to - room ?size - integer
          ?old_room_count_pre ?old_room_count_post
          ?new_room_count_pre ?new_room_count_post - integer)
     (:precondition (and (not (= ?from ?to))
         (> ?old_room_count_pre 0)
         (< ?new_room_count_pre ?size)
         (bag_size ?bx ?size)
         (bagState ?bx ?from ?old_room_count_pre)
         (bagState ?bx ?to ?new_room_count_pre)
         (sum ?old_room_count_pre -1 ?old_room_count_post)
         (sum ?new_room_count_pre 1 ?new_room_count_post)))
     (:effect (and (not (bagState ?bx ?from ?old_room_count_pre))
         (not (bagState ?bx ?to ?new_room_count_pre))
         (bagState ?bx ?to ?new_room_count_post)
         (bagState ?bx ?from ?old_room_count_post))))),
 (:objects bag1 bag2 - boxbag r1 r2 r3 - room),
 (:init ((bagState bag1 r1 4) (bagState bag2 r3 1)),
 (:goal (and (bagState bag1 r2 ?cnt1) (>= ?cnt1 2)
          (bagState bag1 r1 ?cnt2) (>= ?cnt2 1)
          (bagState bag2 r2 ?cnt3) (>= ?cnt3 1))>
```

Algorithm 1 We describe a simplified version of our reformulation algorithm. Reformulate transforms PS into PS' by bagging type t, where type t satisfies the requirements.

```
reformulate(PS : problem space, Bags : partition)

  // Where PS = <T,P,A,O,I,G>,
  // Bags are all the bags of baggable type t
  T' ← T \ {t} ∪ {"tbag"}
  O' ← O \ {o ∈ O : type(o) = t} ∪ {bag_i : 0 < i ≤ |Bags|}
  P_t ← {p ∈ P : a ∈ arguments(p) ∧ type(a) = t}
  At_t // The attributes of type t
  bagState_t ← ("bagState" ?bag - tbag At_t ?cnt - integer)
  P' ← (P \ P_t) ∪ {bagState_t}
  I' ← reformulate-init(I, Bags) // Described below
  G' ← reformulate-goal(G, Bags) // Described below
  A' ← reformulate-actions(A, Bags) // Described below
  PS' ← <T',P',A',O',I',G'>

  return PS'
```

Algorithm Explication *Reformulating Types and Objects:* The reformulation of types and of objects, with respect to the partition, $Bags$, is straightforward. T' is simply T minus type t plus the new type $tbag$, which is simply t's type id concatenated with "*bag*", e.g., $boxbag$ type replaces box. $Bags$ is the partition of all the objects of type t based on the the ISE relation, where $o_1 R_{ISE} o_2$ if they have exactly the same attribute values in the initial state. Each bag has an id and a set of initial state equivalent objects. O' is simply O minus the type t objects plus all the new bag ids of type $tbag$, e.g., objects a, b, c, d are replaced with $bag1$ of type $boxbag$, and e is replaced with $bag2$.

Reformulating Predicates: The reformulation of the predicates is slightly more complicated. The goal is to assemble all of the information about type t objects from the various predicates with type t arguments into one *macropredicate*, bagState, where we have arguments describing: (1) the bag id; (2) the type t attribute information; and (3) the count of how many objects in that bag have those values in the current state. If the original predicates involving box objects were *(in ?bx - box ?rm - room)* and *(colorOf ?clr - color ?bx - box)* then the following predicate would replace them in the reformulated predicates: *(bagState bx - boxbag rm - room clr - color cnt - integer)*. We can do this because type t is indexable, i.e., no original predicate has more than one type t argument. For unary predicates, the attribute type is boolean.

Reformulating the Initial State (reformulate-init): All the predicates in the original initial state with no type t arguments go unaltered into the reformulated state. Predicates which contain an argument of type t are removed from the initial state. Since we are focusing on ISE to define which objects go in a bag, they all have exactly the same attribute values. One grounded macropredicate is added to the reformulated initial state for each bag. Each macropredicate describes (i) the bag id; (ii) the value of each attribute for this bag; and (iii) the number of objects in the bag. For non-unary predicates, if there are no instances of those predicates for a reformulated object, then the bagState argument values for that predicate are "noValue".

Reformulating the Goal (reformulate-goal): The process of reformulating the goal is different as the goal is a partial state. Where there are no original predicates to set their corresponding attribute values, instead of storing *noValue* we store an anonymous variable ("_"). We also leave the count variable in the macropredicates ungrounded, as ($?cnt$ - integer). The goal does not require exactly $?cnt$ bags to have a particular set of attributes, but rather a minimum of $?cnt$. Therefore, for each macropredicate in the reformulated goal, one $\geq$ predicate is added, specifying the minumum count required to satisfy the goal. For example, suppose that *a, b, c, d* and *e* are all in *bag1* and the original goal is {*(in a r1) (in b r1) (color a green) (in c r1) (in d r2)* }. The reformulated goal will be {*(bagState bag1 r1 _ ?cnt*$_1$*), (bagState bag1 r1 green ?cnt*$_2$*), (bagState bag1 r2 _ ?cnt*$_3$*), (*$\geq$*, ?cnt*$_1$*, 3), (*$\geq$ *?cnt*$_2$ *1), (*$\geq$ *?cnt*$_3$ *1)* }. In this example, a is required to be both $green$ and in $r1$, whereas b and c are only required to be in $r1$ and their color is irrelevant. Three macropredicates are required, one describing $\{a\}$, one describing $\{a, b, c\}$ and one describing $\{d\}$. Each macropredicate has a count which must be greater than or equal to the number of objects described by its macropredicate's associated helper predicate. Just like the reformulation of the initial state, predicates with an argument of type t are removed and those without, are unchanged.

Reformulating Actions (reformulate-actions): Reformulating the actions is the most complicated part of the reformulation process. The reformulated actions are responsible for keeping the bagState counts correct. This means that the actions' reformulated preconditions and effects must faithfully reflect the semantics of the original actions. While the reformulated actions do not capture the object ids, they must correctly capture the relevant bag ids.

Our simplifying assumptions of *Only One Bag Type Parameter Modified Per Operator* and *No No-ops* means there is a one-to-one correspondence between the original actions

and the reformulated actions. Operator reformulation must be done so that given an original state and its reformulated version, (i) an original action, α, applies to the original state, $s1$, iff the corresponding reformulated action, α', applies to the reformulated state, $s1'$, and (ii) the result, $s2$, of applying α to $s1$ reformulates to the same reformulated state as the result of applying α' to $s1'$.

This process affects all three parts of an action: parameter list, preconditions and effects (see *Running Example 1*). If the new attribute values differ from the old ones, then the action will modify the bagState. We need to add 4 count fields: the count for the old bagState before the action has been applied and its count after; the count for the new bagState before the action has been applied and its count after [2].

The reformulated parameter list contains all the parameters from the original plus any additional parameters needed to capture attribute/count values. When a reformulated object is modified by an action we need to know all of the old attribute values, all of the new attribute values, and all of the originally unmentioned attribute values. We need these in order to identify the new bagState and the old bagState during solution transformation.

The reformulated preconditions contain 2 new macro-predicates: one describing the old bag attributes before the action has been applied and the other describing its new attributes before the action has been applied. The count of the old one will be decremented by the action, and therefore we need a ">" predicate asserting that this count is greater than 0. The count of the new one will be incremented, and therefore we have a "<" predicate asserting that this count is less than the number of objects in the bag.

The reformulated effects contain the other 2 new macro-predicates: one describing the old bag attributes after the action has been applied and the other describing its new attributes after the action has been applied. Preconditions and effects with an argument of type t are removed and those without are unchanged.

Correctness of Reformulation In order to talk about the correctness of state reformulation, we need to define *bagState refinement*. Given a set of bagState instances for a bag id and the partition $Bags$, the refinement of that set of bagStates is a set of predicates for each object in that bag having the attribute values specfied in the bagStates. For example, given the following set of bagStates, $\{$*(bagState bag1 rm1 green 2)(bagState bag1 r3 red 3)*$\}$, and $Bags$; the partition $\{\{a\ b\ c\ d\ e\}\{f\ g\}\}$ where the equivalence classes are bag1 and bag2, respectively. We refine bag1 into the following set of predicates: $\{$*(in ?b1 r1) (color ?b1 green) (in ?b2 r1) (color ?b2 green) (in ?b3 r3)(color ?b3 red) (in ?b4 r3) (color ?b4 red) (in ?b5 r3)(color ?b5 red)*$\}$ where there is a bijection between the sets $\{$*a, b, c, d, e*$\}$ and $\{$*?b1, ?b2, ?b3, ?b4, ?b5*$\}$, so the given set of bagStates could be refined into any one of the sets dictated by the $\frac{5!}{2!3!}$ possible bijective mappings. The refinement of a bagged state is simply the set of all original predicates which do not have a type t argument, plus the refinement of all the bagState predicates.

Correctness of State Reformulation The reformulation of an original state, s is correct with respect to the partition, $Bags$, if every object of type *t* in s is correctly counted in the corresponding bagState count, and all the other predicates in s are in the reformulated state.

We define the state reformulation s' as being correct if the state s that is being reformulated is one of the possible refinements of s'.

Lemma 1 *Given any complete state description,* s *and partition* Bags *of type* t *objects, our algorithm correctly reformulates, with respect to* Bags*.*

Proof Sketch: This is obvious by inspection of how the reformulated bagStates are constructed.

Correctness of Goal Reformulation The reformulation, g', of goal g is correct with respect to partition $Bags$, if there is a refinement of g' that equals g.

Lemma 2 *Given any goal,* g *and partition* Bags *of type* t *objects, our algorithm correctly reformulates with respect to the partition* Bags*.*

Proof Sketch: This is obvious by inspection of how the reformulated goals are constructed.

Correctness of Action Reformulation An action, a, is correctly reformulated with respect to $Bags$ if (1) when a is applicable to a state, s, then the reformulated action, a', is applicable to the reformulated state, s', and vice versa; (2) when a is applied to state $s1$ then the resulting state, $s2$ reformulates to the same state, $s2'$, as applying a' to $s1'$; (3) when a' is applied to $s1'$, $s2'$ can be refined into $s2$. The 1st condition states that the preconditions must be correctly reformulated, and the 2nd and 3rd state that the effects must be correctly reformulated.

Lemma 3 *Given any original action and the partition,* Bags*, our action reformulation algorithm correctly reformulates the action with respect to* Bags*.*

Proof Sketch: The reformulated action needs to affect the unreformulated parts of the reformulated state exactly the same as it does in the orignal representation. Inspecting the algorithm shows this to be true as they are just passed through unaltered. The reformulated parts need to do two things. One is to effect the change to the reformulated objects exactly as the original action would. This means that the modified object's old attributes must correctly reflect their values as they would for the original action and that the new attributes correctly reflect their values as they would in the original action. This can be seen by inspecting how these aspects of the modified object are computed.

The second thing the reformulation must do is to correctly adjust the counts for the modified object. The preconditions check that there must be at least one bagged object that satisfies the modified object's old attribute values and that the count associated with the old bagState is decremented by 1, and the count associated with new bagState is incremented by 1 and never exceeds the number of objects in that bag. Since we have assumed that at most 1 reformulated object

[2] If the action does not modify the bag (for example, an action which requires a spare hand but the hand remains spare in the effects), then only 1 count is necessary.

can be modified by an action, and since our *No No-ops* assumption ensures that, if a reformulated object is mentioned in the action's effects then it actually does make a modification to the object, this updating of counts is correct.

2) Solving the Problem We assume we use the same sound planner for the reformulated problem space as we would for the original problem space. Therefore, if it solves the bagged problem then its solution is valid in the bagged problem space.

3) Transformation of Solution Our reformulation produces an abstraction of the original problem space, which has lost information. A reformulated state can be refined into a set of states. Consequently, our reformulated solution path can be refined into a set of plans, where at least one plan is a solution to the original problem. **Transformation** is the refinement of the bagged solution into an original space solution (i.e. a plan which satisfies the original space goal).

So far we have shown that: (i) the initial state bag counts are correct; (ii) each bag count has stored every bit of information about each bagged object except for its object id; and (iii) the bagged preconditions and effects correctly reflect the semantics of the original actions. This means that if a bagged operator applies to a bagged state then the operator it was derived from will also apply to any refinement of that bagged state; and that the application of that operator to the refined state will produce a state which will, when reformulated, be the same as the bagged state produced by applying that bagged operator to the bagged state. In this section we show that Baggy is guaranteed to transform a valid solution in the bagged space into a sound solution in the original space. This transformation occurs in 3 steps, Baggy[3] (3a) refines the bagged solution, P', into a valid sequence of original space actions, p; (3b) creates a correction mapping between objects in that final state associated with this sequence to objects in the original goal; and (3c) applies this mapping to p to produce a solution, P, to the original problem.

3a) Refinement of Bagged Solution into a Plan

Running Example 3a Suppose that we have a sequence of bagged actions that change the bagged initial state into a bagged goal state. Below we show a reformulated solution where a bagged state precedes and follows each of the 3 grounded bagged actions. Below the bagged solution we show the original space plan it is refined into.

```
SOLUTION PHASE 1: REFINE OPERATOR SEQUENCE
bag1 = {a, b, c, d}   bag2 = {e}
bagged init, I' =  (bagState bag1 r1 4)(bagState bag2 r3 1)
bagged goal, G' = (and (bagState bag1 r2 ?cnt1) (≥ ?cnt1 2)
                       (bagState bag1 r1 ?cnt2) (≥ ?cnt2 1)
                       (bagState bag2 r2 ?cnt3) (≥ ?cnt3 1))
Bagged solution, P':
 (bagState bag1 r1 4) (bagState bag2 r3 1)
mv_box(bag1 r1 r2 4 3 0 1)
 (bagState bag1 r1 3) (bagState bag1 r2 1) (bagState bag2 r3 1)
mv_box(bag1 r1 r2 3 2 1 2)
 (bagState bag1 r1 2) (bagState bag1 r2 2) (bagState bag2 r3 1)
mv_box(bag2 r3 r2 1 0 0 1)
 (bagState bag1 r1 2) (bagState bag1 r2 2) (bagState bag2 r2 1)


original init, I = (in a r1)(in b r1)(in c r1)(in d r1)(in e r3)
```

[3] For GE bags, Baggy only needs the 1st step.

```
original goal, G = (and(in b r2)(in c r2)(in d r1)(in e r2))

p ← refine-plan(P', I, {bag1, bag2})
Original plan, p:
 (in a r1) (in b r1) (in c r1) (in d r1) (in e r3)
mv_box(a r1 r2)
 (in a r2) (in b r1) (in c r1) (in d r1) (in e r3)
mv_box(b r1 r2)
 (in a r2) (in b r2) (in c r1) (in d r1) (in e r3)
mv_box(e r3 r2)
 (in a r2) (in b r2) (in c r1) (in d r1) (in e r2)
```

Algorithm 3a The algorithm that refines the bagged solution into a plan in the original space is shown below.

```
refine-plan(P': reformPlan, I: state, Bags : partition)
  p ←  < > // Original space empty plan
  S ← I    // Current original space state

  while P' ≠ < >
      α'<v1, ..., vm'> ← pop(P') // Grounded bagged operator
      α <v1, ..., vm> ← refine-action(α'<v1, ..., vm'>, S, Bags)
          // α is any valid original space refinement of α'
          // with the bag parameters grounded to baggable objects
      S ← α<v1, ..., vm>(S) // Update the state by applying α
      p ← p + α<v1, ..., vm>
  return p
```

The Refined Plan is Valid

Lemma 4 *The refinement of a valid* n *step plan, P', in the reformulated problem space by refine-plan produces a valid* n *step plan, p, in the original problem space. The final state, f, resulting from executing p in the initial state, I, is a refinement of the bagged final state, f', resulting from executing P' in the bagged initial state, I'.*

Proof Sketch:

Assume that $Bags$ is the partition of the initial state I using type t. We refer to this partition everywhere in this proof when we talk about reformulating and/or refining. The proof is by induction on the length of the bagged solution.

Base Case: Empty bagged plan. The empty bagged plan would be refined to the empty plan in the original space. The empty plan is always valid. Both of these plans have the same length, zero. The result of applying the empty plan to I is I. By Lemma 1, this is a refinement of the bagged initial state.

Inductive Hypothesis: Assume all bagged plans of length n are transformed into valid plans of n original actions where the final state is a refinement of the final bagged state.

Suppose that bagged plan P' is of length $n+1$. Take the first n steps of P', call it P'_n. By the inductive hypothesis we can refine P'_n into p_n and the final state s_n resulting from executing p_n in I is a refinement of the bagged state, s'_n resulting from executing P'_n in I'.

Lemma 3 means since the last bagged step, α', is applicable to the state s'_n then α is applicable to s_n and the state, s_{n+1}, resulting from that application is a refinement of the state, s'_{n+1} resulting from applying α' to s'_n. The transformed plan, p, has length $n+1$.

3b) Computing Correction Mapping

Running Example 3b While the new plan, p, can be validly applied to the original initial state, it does not necessarily result in a final state, f, which satisfies the original goal, G. The correction mapping for our running example is shown below. $(x, y) \leftrightarrow$ object x in f maps to object y in G.

```
SOLUTION PHASE 2: COMPUTE CORRECTION MAPPING
bag1 = {a, b, c, d}   bag2 = {e}
goal, G = (and (in b r2)(in c r2)(in d r1)(in e r2))
final state, f = (in a r2)(in b r2)(in c r1)(in d r1)(in e r2)

M ← compute-correction-mapping(bag1, G, f)
  ∪ compute-correction-mapping(bag2, G, f)
Correction mapping, M:
{(a, c), (b, b), (c, a), (d, d), (e, e)}
```

Algorithm 3b We compute a bijective correction mapping, M, which maps objects to other objects in the bag so that applying M satisfies the original goal G.

```
compute-correction-mapping(Bag : bag, G: state, f : state)
  M ← {} // Correction mapping
   ∀ obj1 ∈ Bag
    f1 ⊆ f // Predicates in f which have obj1 as an argument
    ∀ obj2 ∈ Bag
      G2 ⊆ G // Predicates in G which have obj2 as an argument

      // If replacing obj1 with obj2 in f1 satisfies G2
      if G2 ⊆ replace(obj1, obj2,  f1)

        // Then obj1 can be mapped to obj2
        M ← M ∪ (obj1, obj2)

  M ← prune-mappings(M) // Prune M such that ∀ b ∈ Bag ∃ precisely
      // one map (x,b) and one map (b,y) for some x,y ∈ Bag

  return M
```

Mapping Final State to Goal State We introduce the function *correct-state(M, $State$)* which uses correction mapping M to return a corrected version of $State$.

Lemma 5 *The correction mapping,* M*, constructed by compute-correction-mapping, is constructed such that it is guaranteed that correct-state(M, f) satisfies* G.

Proof Sketch: Our bijection is from each bag to itself. Since the type is baggable it must be action equivalent, i.e., that the actions only check for attribute values not for object id. This means that any such bijection will leave the action sequence valid and also means it will end in a final state that satisfies the goal.

3c) Applying Correction Mapping to Obtain Solution

Running Example 3c

```
SOLUTION PHASE 3: APPLY CORRECTION MAPPING TO OPERATION SEQUENCE
M = {(a, c), (b, b), (c, a), (d, d), (e, e)} // Correction mapping
p =
<mv_box(a r1 r2), mv_box(b r1 r2), mv_box(e r3 r2)>

P ← correct-plan(M, p)
Original space solution, P:
<mv_box(c r1 r2), mv_box(b r1 r2), mv_box(e r3 r2)>
```

Algorithm 3c

```
correct-plan(M : correction mapping, p : plan)
  P ← < > // Empty solution
  ∀ step ∈ p
    STEP ← correct-action(M, step) // Apply the correction mapping
    P ← P + <STEP>
  return P
```

Applying Correction Mapping to Plan Gives Solution Let P be the sequence of actions achieved by calling *correct-plan(M, p)*.

Lemma 6 *The state resulting from applying P to I, satisfies G.*

Proof Sketch: Let f be the state resulting from applying p to I. Since all the bag type t objects are action equivalent and all the objects in the same bag have the same initial state attribute values then if we take two objects, x and y, from the same bag and consistently map x to y and y to x in every step in p then the final state will be exactly the same as f except that wherever x appeared in f, y will appear in the new final state and everywhere y appeared in f, x will appear in the new final state. This is also obviously true if we extend that mapping between two objects to the bijective mapping, M, constructed by *compute-correction-mapping*. Applying this correction mapping to p and applying it to I will lead to a state which satisfies G. Thus, P is a solution to the original problem.

Theorem 1 *The process of reformulating the problem, solving that problem with a sound planner, and transforming that reformulated solution back into the original representation is sound.*

Proof Sketch: The validity of this theorem derives from the proof sketches of the correctness of the reformulation and the proof sketches concerning this transformation process.

Experimental Results

We compare our system to a state-of-the-art symmetry reduction system, Metis (Alkhazraji et al. 2014) which has three main components built on top of Fast Downward (Helmert 2006); an incremental LM-cut heuristic, symmetry reduction using Orbit search (Domshlak, Katz, and Shleyfman 2015), and partial order reduction with strong stubborn sets. In Table 1 we isolate the symmetry reduction component and disable the other two components. All the results in Table 1 were run with Metis' Fast Downward running A* with the Blind heuristic. It shows the 11 domains Baggy can reformulate from the IPC-11/14 competitions: Barman-opt11 (B1), Elevators-opt11 (E1), Floortile-opt11 (F1), Nomystery-opt11 (N1), Transport-opt11 (T1), Barman-opt14 (B4), Childsnack-opt14 (C4), Floortile-opt14 (F4), Hiking-opt14 (H4) Tetris-opt14 (Te4), Transport-opt14 (T4). We include a new domain (N2), discussed later. *SA* denotes the largest number of problems solved by any one of the settings over the total number of reformulatable problems from the domain. Under each setting we report *s*, the number of problems solved and *E*, defined as $E = E^*/E_s$ where E_s is the number of nodes expanded by this setting and E^* is the minimum nodes expanded by any setting. If a setting expanded the fewest nodes it will receive a 1 for that problem and if it expanded twice as many nodes it would receive 0.5. These values are summed across all the problems within a domain, so if 20 domain problems were solved then a perfect *E* value would be 20. This is the same formula used in (Domshlak, Katz, and Shleyfman 2012). The last two columns show the reduction factor for the reformulation and orbit search respectively. This is the number of times the search space has been reduced for each compared to blind search in the original space. (A 2 means the original blind space is twice the size). If no problem is solved in either

Table 1: Summarized results for s, and E of 2 representations using Blind Search with and without Symmetry breaking

Do	SA	Reformulated Blind BR		Reformulated Orbit Srch BOSR		Original Blind BO		Original Orbit Srch BOSO		Reduction Factor Ref	Reduction Factor Orb
		s	E	s	E	s	E	s	E		
B1	12/20	**12**	5.38	**12**	**12**	4	0.07	8	0.42	**36.**	4.5
E1	8/15	**8**	7.20	**8**	**8**	6	2.94	**8**	5.40	**1.8**	1.2
F1	2/20	**2**	1.04	**2**	**2**	**2**	0.29	**2**	0.93	**3.6**	3.2
N1	9/20	**9**	**9**	**9**	**9**	8	4.94	**9**	8.57	**3.7**	**3.7**
T1	6/19	**6**	5.15	**6**	**6**	**6**	3.00	**6**	4.72	**1.5**	**1.5**
B4	8/14	6	1.72	**8**	**8**	0	0	3	0.06	>**192.**	>12.
C4	8/20	6	1.16	**10**	**10**	0	0	6	4.46	>3430.	>**8790.**
F4	0/20	0	0	0	0	0	0	0	0	>1.3	>**1.5**
H4	17/20	14	4.59	**17**	**17**	11	1.89	**17**	**17**	5.0	**27**
Te4	8/14	7	5.71	**8**	**8**	4	1.56	**8**	**8**	2.13	**6.07**
T4	6/14	**6**	5.52	**6**	**6**	**6**	3.36	**6**	5.49	1.8	**2.6**
N2	3/6	**3**	**3**	**3**	**3**	2	0.9	2	0.9	**9.9**	1
Tot	87/203	79	49.47	**89**	**89**	49	18.95	75	55.95		

space, the last common f-bound is used to calculate a lower bound. Throughout this analysis we use number of nodes expanded since last jump to normalize for different tree orderings on the last level. We used the 4GB memory restriction from IPC, but we increased the time to an hour. Baggy itself is very fast - the maximum reformulation time was 316s and the maximum solution transformation time was 0.15s. The median times were 0.54s and 0.15s respectively. For these experiments, we used both ISE and GE relations to produce bags.

Table 2: Summarized results for s, and E of 2 representations with 3 state-of-the-art planners

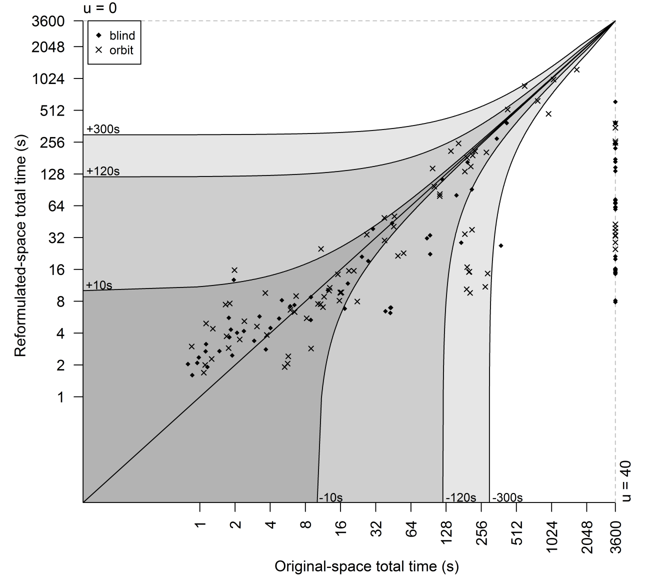
Do	SA	Reformulated Sym	Reformulated Metis		Reformulated iPDB		Original Sym	Original Metis		Original iPDB	
		s	s	E	s	E	s	s	E	s	E
B1	20/20	**20**	8	8	12	**6.4**	11	8	1.3	4	0
E1	14/15	**14**	**14**	**10.8**	12	0.5	**14**	13	9.9	12	2.1
F1	14/20	12	9	**9**	3	0	**14**	9	4.2	2	0
N1	20/20	14	13	0.4	18	6.4	16	17	**12.7**	**20**	9.8
T1	9/19	**9**	7	4.7	6	0.1	**9**	7	**6.2**	8	1.3
B4	10/14	**10**	3	3	6	**3.6**	6	3	0.3	0	0
C4	13/20	**13**	4	3.9	6	0.6	3	8	**7.9**	0	0
F4	20/20	12	8	**8**	2	0.0	**20**	8	3.1	0	0
H4	19/20	14	13	**11.1**	17	7.8	**19**	14	7.4	13	3.2
Te4	11/14	**11**	7	5.4	3	1.4	8	8	**7**	1	1
T4	7/14	**7**	6	**5.0**	6	1.3	**7**	5	4.7	6	1.2
N2	6/6	6	4	0.0	5	1.1	4	3	4	4	1.6
Tot	163/203	**142**	96	**69.3**	96	29.2	131	103	68.7	70	20.2

Table 1 shows that the blind heuristic on the reformulated domain (BR) always solves at least as many problems as the blind heuristic on the original domain (BO). In addition the E value is almost always much larger for BR than BO (the totals are 49.47 versus 18.95). The reformulated state space is always smaller and usually a lot smaller.

BR and blind orbit search on the original representation (BOSO) solve the same number of problems in 7 domains. BOSO solves 3 more problems in the Hiking domain and 1 more problem in the Tetris domain. BR solves 4 more problems in Barman11 and 3 more problems in Barman14. The E values tell a more important story. In all domains except Childsnack, Hiking and Tetris, BR has a smaller search space, often significantly smaller. This is surprising as these domains are not ideal for our technique. We perform worse in Childsnack because we cannot bag *child* as it is not single-valued[4]. In some domains we do better than BOSO because some of the problems are ISE baggable and therefore we find symmetries which BOSO cannot find.

[4]The *serve_sandwich* operator has a precondition of *(waiting ?c - child,?p - place)*, which is not removed in the effects when *(served ?c - child)* is added. We have techniques for improving this, by running a PDDL-repair process before Baggy, but this is not included in this paper.

Figure 1: Total planner time for each reformulatable problem which was solved by the planner (from Table 1). Contours indicate the difference in running time between reformulated and original space. Problems which were solved in one search space but not the other are along the dotted line (at 3600s). The u value counts the number of points along the unsolved line.



One of the advantages of reformulating the lifted PDDL as preprocessing, rather than doing symmetry breaking during the search process itself, is that you can run any PDDL-based planner on it. We also ran Orbit Search on the reformulated representation (BOSR), to check whether it finds extra symmetries that Baggy does not (this is why the E value is the same as the s value). In terms of the E-value, BOSR never does worse and often does better than either BR or BOSO. With the exception of N1 and N2, BOSO always finds further symmetries not exploited by Baggy. For instance Orbit search can find more complicated symmetries then object symmetries

In Table 2 we run 3 state-of-the-art planners on both the original representation and the reformulated representation. These are SymBA*[5] (Torralba et al. 2014), Metis (Alkhazraji et al. 2014), and iPDB (Haslum et al. 2007). SymBA* (Sym) and iPDB both solve more problems in the reformulated representation than in the original representation. Metis solves fewer problems in the reformulated space, this is because the delete relaxation that is the basis of LM-cut is not an appropriate heuristic for a bagged representation. Once you pick up one "ball" you can put it down as many times as you wish. So heuristics based on delete relaxation are very inaccurate in the bagged representation. iPDB is not as disadvantaged by the reformulated representation,

[5]There is no E given for the SymBA* system since it does not represent nodes in the same way as Fast Downward, there is no way to directly compare the nodes expanded .

but there are still domains where the original representation does better. This is probably caused by the number of SAS+ variables and operators. You can see in Table 3 that sometimes the reformulated representation has fewer variables and operators, and sometimes there are more. In addition you will notice that you frequently get fewer variables and operators if you use the SymBA* Translator and Preprocessor, but these experiments were all done using the regular FD Translator and Preprocessssor (except when SymBA* was run). Fewer variables is often an advantage for iPDB.

Figure 1 shows the times for all the problems in Table 1. You can see that there is a time penalty for reformulation on the smaller problems. But for the bigger problems the reformulated problems are frequently faster. Bear in mind, since this is a log graph, the win for the bigger problems is much larger than the loss for the smaller problems. In addition, the numbers of problems which the reformulated representation solved and the original representation didn't is much higher than the inverse number. A similar graph for Table 2 would not be as clear cut. The reformulated representation is obviously confusing for traditional heuristics. This is an opportunity for the community to develop heuristics which work well on non-standard representations.

We created a new set of nomystery problems, N2. These use the same domain file as the regular problems, but all the packages start in location l0 and are delivered each to its own location. This domain should show the biggest difference between Baggy and Orbit Search, since there should be no GE. BOSR and BR perform the same since there are no additional symmetries to remove. Both do better than BO and BOSO. With the state-of-the-art planners, they all solve more problems in the reformulated representation, but Metis again performs worse on the reformulated representation than the others because of its delete relaxation based heuristic. This shows the real power of Baggy over other symmetry reduction techniques. Since in the original nomystery domain the new representation performed worse, it has become obvious that its performance is based upon the specific problems chosen and the size of the bags created, as opposed to something that can be determined at the domain level. This highlights the need to chose a heuristic/representation combination on a problem by problem basis Notice that none of the planners got anywhere close to the 163 problems solved overall (see Tot/SA cell of Table 2). A system like RIDA* (Barley, Franco, and Riddle 2014) should be used to decide when to use which representation/heuristic combination.

Table 3: Largest number of SAS variables and operators for each representation and translator and preprocessor pair. The last column contains the size of the largest bag.

Domain	Reformulation		Reform-SymBA*		Original		Original-SymBA*		Bag
	Vars	Ops	Vars	Ops	Vars	Ops	Vars	Ops	Size
B1	95	49400	67	25691	162	1016	162	908	9
E1	93	1296	93	1296	16	1008	16	1008	3
F1	162	4040	162	482	48	1104	45	540	3
N1	86	9106	86	8170	14	8890	14	7954	5
T1	132	4536	132	4536	15	3408	15	3408	3
B4	104	60920	72	32891	172	1078	172	970	9
C4	35	442971	34	26992	48	1025	48	1025	15
F4	135	1146	135	397	39	476	37	296	2
H4	58	27072	58	27072	14	8200	14	8200	4
Te4	2020	32720	320	788	1982	32588	282	920	4
T4	384	12186	384	12186	13	10746	13	10746	2
N2	16	6564	16	6564	14	4848	14	4848	12

Related Research

There has been considerable work on problem reformulation, starting with George Polya's *How to Solve It* (1957). Due to lack of space we will focus on planning-specific reformulation research. The Fast Downward system (Helmert 2006) first transforms the PDDL representation into a multi-valued planning task, similar in spirit to SAS^+. Using this representation, the system generates four internal data structures, which it uses to search for a plan. Helmert (Helmert 2009), extending this work, focused on turning PDDL into a concise grounded representation of SAS^+. Additional work in this area transforms PDDL into binary decision diagrams (Haslum 2007), transforms between PDDL and causal graphs (Helmert 2006), and identifies and removes irrelevant parts from a problem representation (Haslum, Helmert, and Jonsson 2013).

As noted earlier, our system has much in common with symmetry reduction systems. Fox and Long (1999) group symmetric objects together in TIM. They require objects to be indistinguishable in both the initial state and the goal description. They keep track of the symmetry groups during planning but only with respect to the goal description, so they cannot remove all the symmetries in gripper. They have a constraint similar to our action equivalence, which tests if objects are used as constants within actions. Pochter et. al. (2011) generalize the work by Fox and Long, by using generators to create automorphic groups. These groups are based on SAS^+ and so are more general than objects. They still require the symmetric groups to be indistinguishable in both the initial state and goal description. Domshlak et. al. (2012) extended this work to only require symmetric groups to be indistinguishable in the goal description in the DKS system. They compared their work to Pochter's system, where they solved 8 more problems over 30 domains. Metis (Alkhazraji et al. 2014), uses orbit search to do symmetry breaking. It is an improvement on DKS, since it does not store extra information in each state. Metis also includes an incremental LM-cut heuristic and partial order reduction with strong stubborn sets.

The closest work to our automated system for creating these transformations is the system by de la Rosa et. al. (2015). They reformulate PDDL into PDDL and they merge objects in a similar way. The main differences are 1) we merge objects if they are the same in the initial state or the same in the goal description whereas their system merges objects if they do not appear in the goal description 2) they explicitly use numeric fluents in their modeling, restricting them to planners that support them such as metric-FF (Hoffmann 2003) 3) both our systems transform the solution back to the original representation but our system can generate plans which have different explicit values specified in the goal state. In addition recent work by Bartak et. al. (Barták, Dovier, and Zhou 2015; Zhou, Bartak, and Dovier 2015; Bartak and Vodrazka 2015) has looked at representing planning problems in the PICAT planner module for tabled logic programming. They have shown that representing the planning task as a structured state representation leads to increased problem solving performance. Our bagState predicates are related to this structured state representation.

Concluding Remarks

We have shown that one can solve more problems and expand fewer nodes, at least in some domains, by spending just a little time transforming the lifted PDDL representation. When we create a new state space, it is always smaller than the original space. Our transformation involves grouping into bags, objects that are indistinguishable with respect to the actions and the initial state or the goal description. By doing so, we only keep track of their counts instead of identifying them individually, and we avoid unnecessary combinatorial explosion. Since the transformations produce PDDL files, they can be used with any planning system. Our experiments show that, in some domains, we reduce the state space more than state-of-the-art symmetry reduction techniques.

This paper had 3 main contributions. We presented Baggy and showed it is a sound method to transform PDDL, which always results in a smaller state space. We showed that there are situations where a state-of-the-art heuristic problem solver can perform better in the reformulated representation than in the original representation. Lastly we showed that this is not always the case, for some problem/planner combinations, the new representation can perform worse.

These encouraging results suggest many directions for future work. Our new representation does not work well with all heuristics (especially delete-relaxation based ones) so we planning to us a RIDA* type system to choose an appropriate representation/heuristic combination on a problem by problem basis. We are currently creating heuristics which work better on Bagged representations. One of our main areas of future research is to look at planners that do not used grounded SAS^+ representations, such as metric-ff (Hoffmann 2003) or SAT-based planners (Davies et al. 2015). We have found that the compactness of the SAS^+ representation is very sensitive to the specific PDDL formulation. We will be exploring this space of equivalent PDDL representations in order to make the PDDL we create more efficient in terms of its SAS^+ representation. We have develop several techniques for optimizing the PDDL we create to make efficient SAS+ representations.

Acknowledgements

This material is based upon work supported by the Air Force Office of Scientific Research, Asian Office of Aerospace Research and Development (AOARD) under award number FA2386-15-1-4069. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force. Financial support for this research was in part provided by Brazil's CAPES (Science Without Borders). Special thanks to Rob Holte and Alvaro Torralba for comments on earlier versions of this paper.

References

Alkhazraji, Y.; Katz, M.; Mattmuller, R.; Pommerening, F.; Shleyfman, A.; and Wehrle, M. 2014. Metis: Arming fast downward with pruning and incremental computation. In *The 2014 ICAPS - Description of Planners*.

Amarel, S. 1971. Representations and modeling in problems of program formation. *Machine Intelligence 6*.

Bäckström, C., and Nebel, B. 1995. Complexity results for sas+ planning. *Computational Intelligence* 11(4):625–655.

Barley, M.; Franco, S.; and Riddle, P. 2014. Overcoming the utility problem in heuristic generation: Why time matters. In *ICAPS*.

Bartak, R., and Vodrazka, J. 2015. The effect of domain modeling on efficiency of planning: Lessons from the nomystery domain. In *TAAI*.

Barták, R.; Dovier, A.; and Zhou, N.-F. 2015. On modeling planning problems in tabled logic programming. In *PPDP*.

Davies, T.; Pearce, A. R.; Stuckey, P.; and Lipovetzky, N. 2015. Sequencing operator counts. In *ICAPS*.

de la Rosa, T., and Fuentetaja, R. 2015. Automatic compilation of objects to counters in automatic planning. case of study: Creation planning.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2012. Enhanced symmetry breaking in cost-optimal planning as forward search. In *ICAPS*.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2015. Symmetry breaking in deterministic planning as forward search: Orbit space search algorithm. Technical Report IS/IE-2015-02, Israel Institute of Technology.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *IJCAI*, 956–961.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI*.

Haslum, P.; Helmert, M.; and Jonsson, A. 2013. Safe, strong and tractable relevance analysis for planning. In *ICAPS*.

Haslum, P. 2007. Reducing accidental complexity in planning problems. In *IJCAI*, 1898–1903.

Helmert, M. 2006. The fast downward planning system. *JAIR* 26(1):191–246.

Helmert, M. 2008. *Understanding planning tasks: domain complexity and heuristic decomposition*.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5).

Hoffmann, J. 2003. The metric-ff planning system: Translating"ignoring delete lists"to numeric state variables. *JAIR*.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2011. Exploiting problem symmetries in state-based planners. In *AAAI*.

Pólya, G. 1957. *How to solve it: A new aspect of mathematical method*. Princeton University Press, second edition.

Riddle, P.; Barley, M.; and Franco, S. 2015. Bagged representations in PDDL. In *IPC Workshop*.

Riddle, P.; Holte, R.; and Barley, M. 2011. Does representation matter in the planning competition? In *SARA*.

Torralba, Á.; Alcázar, V.; Borrajo, D.; Kissmann, P.; and Edelkamp, S. 2014. SymBA*: A symbolic bidirectional A* planner. In *The 2014 ICAPS - Description of Planners*.

Zhou, N.-F.; Bartak, R.; and Dovier, A. 2015. Planning as tabled logic programming. *Theory and Practice of Logic Programming* 15(4-5):543–558.

On State-Dominance Criteria in Fork-Decoupled Search

Álvaro Torralba and **Daniel Gnad** and **Patrick Dubbert** and **Jörg Hoffmann**
Saarland University
Saarbrücken, Germany
{torralba, gnad, hoffmann}@cs.uni-saarland.de; s9padubb@stud.uni-saarland.de

Abstract

Fork-decoupled search is a recent approach to classical planning that exploits *fork* structures, where a single *center* component provides preconditions for several *leaf* components. The *decoupled states* in this search consist of a center state, along with a *price* for every leaf state. Given this, when does one decoupled state dominate another? Such *state-dominance* criteria can be used to prune dominated search states. Prior work has devised only a trivial criterion. We devise several more powerful criteria, show that they preserve optimality, and establish their interrelations. We show that they can yield exponential reductions. Experiments on IPC benchmarks attest to the possible practical benefits.

Introduction

Fork-decoupled search is a new approach to state-space decomposition in classical planning, recently introduced by Gnad and Hoffmann (2015). The approach partitions the state variables into disjoint subsets, *factors*, like in factored planning (e. g. (Amir and Engelhardt 2003; Kelareva *et al.* 2007; Fabre *et al.* 2010; Brafman and Domshlak 2013)). While factored planning is traditionally designed to handle arbitrary cross-factor interactions, fork-decoupling assumes these interactions to take a fork structure (Katz and Domshlak 2008; Katz and Keyder 2012; **?**), where a single *center* provides preconditions for several *leaves*. A simple pre-process can determine whether such a fork structure exists, and extract a corresponding factoring if so.

Fork factorings identify a form of "conditional independence" between the leaf factors: Given a fixed center path π^C, the *compliant* leaf moves – those leaf moves enabled by the preconditions supplied along π^C – can be selected independently for each leaf. The decoupled search thus searches only over center paths π^C. Each *decoupled state* in the search represents the compliant leaf moves in terms of a *pricing function*, mapping each leaf-factor state s^L to the cost of a cheapest π^C-compliant path achieving s^L. As Gnad and Hoffmann (henceforth: GH) show, this can exponentially reduce state space size. It may also cause exponential blow-ups though.

The worst-case exponential blow-ups result from irrelevant distinctions in pricing functions. One means to combat this, and more generally to improve search, is *dominance pruning*, pruning a state $s^\mathcal{F}$ if a better state $t^\mathcal{F}$ has already been seen. But, given the complex structure of decoupled states, when is one "better" than another? GH employ the trivial criterion, where $s^\mathcal{F}$ and $t^\mathcal{F}$ must have the same center state and $t^\mathcal{F}$ needs to have cheaper prices than $s^\mathcal{F}$ for all leaf states. Here we introduce advanced methods, analyzing the structure of decoupled states to identify (and then, disregard) irrelevant distinctions. We devise several such methods, using different sources of information. We show that the methods preserve optimality, and we characterize their relative pruning power. We show that they can yield exponential search reductions. Experiments on International Planning Competition (IPC) benchmarks attest to the possible practical benefits.

For space reasons, we can only outline our proof arguments. The full proofs are available in an online TR (Torralba *et al.* 2016).

Background

We use finite-domain state variables (Bäckström and Nebel 1995; Helmert 2006). A *planning task* is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, I, G \rangle$. $\mathcal{V}$ is a set of *variables*, each associated with a finite domain $\mathcal{D}(v)$. I is the *initial state*. The *goal* G is a partial assignment to $\mathcal{V}$. $\mathcal{A}$ is a finite set of *actions*, each a triple $\langle \mathsf{pre}(a), \mathsf{eff}(a), \mathsf{cost}(a) \rangle$ of *precondition*, *effect*, and *cost*, where $\mathsf{pre}(a)$ and $\mathsf{eff}(a)$ are partial assignments to $\mathcal{V}$, and $\mathsf{cost}(a) \in \mathbb{R}^{0+}$. For a partial assignment p, we denote with $\mathcal{V}(p) \subseteq \mathcal{V}$ the subset of variables on which p is defined. For $V \subseteq \mathcal{V}(p)$, we denote with $p[V]$ the assignment to V made by p. We identify (partial) variable assignments as sets of variable/value pairs, written as (var, val). A *state* is a complete assignment to $\mathcal{V}$. Action a is *applicable* in state s if $\mathsf{pre}(a) \subseteq s$. Applying a in s changes the value of all $v \in \mathcal{V}(\mathsf{eff}(a))$ to $\mathsf{eff}(a)[v]$, and leaves s unchanged elsewhere. We will sometimes write $s \xrightarrow{a} t$ for a transition from s to t with action a. A *plan* for Π is an action sequence π iteratively applicable in I which results in a state s_G where $G \subseteq s_G$. The plan is *optimal* if its summed-up cost, denoted $\mathsf{cost}(\pi)$, is minimal among all plans for Π.

We next give a recap of GH's definitions. A *fork factoring* $\mathcal{F}$ is a partition of $\mathcal{V}$ identifying a fork structure. Namely, (i) every action $a \in \mathcal{A}$ affects (touches in its effect) exactly one element (*factor*) of $\mathcal{F}$, which we denote $F(a)$. And (ii) there is a *center* $F^C \in \mathcal{F}$ s.t., for every

$a \in \mathcal{A}$, $\mathcal{V}(\mathsf{pre}(a)) \subseteq F^C \cup F(a)$. We refer to the factors $F^L \in \mathcal{F}^L := \mathcal{F} \setminus \{F^C\}$ as *leaves*. We refer to actions affecting F^C as *center actions*, and to actions affecting a leaf as *leaf actions*. By construction (each action affects only one factor) these two kinds of actions are disjoint. Center actions are preconditioned only on F^C, leaf actions may be preconditioned on F^C and the leaf they affect. In brief: *the center provides preconditions for the leaves, and there are no other cross-factor interactions.*

As a running example, we use a Logistics-style planning task with a truck variable t, a package variable p, and n locations $l_1, \dots, l_n$. $I = \{(t, l_1), (p, l_1)\}$ and $G = \{(p, l_2)\}$. Action *drive*(x, y) moves the truck from any location x to any other location y. The package can be loaded/unloaded at any location x with actions *load*(x)/*unload*(x) respectively. Then $\mathcal{F} = \{\{t\}, \{p\}\}$ is a fork factoring where $\{t\}$ is the center and $\{p\}$ is the single leaf. If we have m packages p_i, we can set each $\{p_i\}$ as a leaf.

Not every task Π has a fork factoring. GH analyze Π's causal graph (e. g. (Knoblock 1994; Jonsson and Bäckström 1995; Brafman and Domshlak 2003; Helmert 2006)) in a pre-process, identifying a fork factoring if one exists, else *abstaining* from solving Π. We follow this approach here. In what follows, we assume a fork factoring $\mathcal{F}$. Variable assignments to F^C are called *center states*, and for each $F^L \in \mathcal{F}^L$ assignments to F^L are *leaf states*. We denote by S^L the set of all leaf states, across $F^L \in \mathcal{F}^L$. For each leaf, s^L_I denotes the initial leaf state. For simplicity (wlog), we will assume that every leaf has a single goal leaf state, s^L_G.

Decoupled search searches over sequences of center actions π^C, called *center paths*, that are applicable to I. For each π^C, it maintains a compact representation of the *leaf paths* π^L that *comply* with π^C. A leaf path is a sequence of leaf actions applicable to I when ignoring preconditions on F^C. Intuitively, given the fork structure, a fixed center path determines what each leaf can do (independently of all other leaves, as they interact only via the center). This is captured by the notion of compliance: π^L complies with π^C if it uses only the center preconditions supplied along π^C, i. e., if π^L can be scheduled alongside π^C s.t. the combined action sequence is applicable in I. Decoupled search goes forward from I until it finds a center path π^C to a center goal state where every leaf has a π^C-compliant leaf path π^L to its goal leaf state. The global plan then results from augmenting π^C with the paths π^L.

In detail: A *decoupled state* $s^\mathcal{F}$ is given by a center path $cp(s^\mathcal{F})$. Its center state $cs(s^\mathcal{F})$ and *pricing function* $prices(s^\mathcal{F}) : S^L \mapsto \mathbb{R}^{0+}$ are induced by $cp(s^\mathcal{F})$, as follows. $cs(s^\mathcal{F})$ is the outcome of applying $cp(s^\mathcal{F})$ to s^L_I. $prices(s^\mathcal{F})$ maps each leaf state s^L to the cost of a cheapest $cp(s^\mathcal{F})$-compliant leaf path ending in s^L (or ∞ if no such path exists).[1] The *initial decoupled state* $I^\mathcal{F}$ has the empty center path $cp(I^\mathcal{F}) = \langle\rangle$. A *goal decoupled state* $s^\mathcal{F}_G$ is one with a *goal center state* $cs(s^\mathcal{F}_G) \supseteq G[F^C]$ and where, for every leaf factor $F^L \in \mathcal{F}^L$, its goal leaf state s^L_G has been reached, i. e., $prices(s^\mathcal{F}_G)[s^L_G] < \infty$. The actions applicable in $s^\mathcal{F}$ are those center actions a where $\mathsf{pre}(a) \subseteq cs(s^\mathcal{F})$. Applying a to $s^\mathcal{F}$ results in $t^\mathcal{F}$ where $cp(t^\mathcal{F}) := cp(s^\mathcal{F}) \circ \langle a \rangle$, inducing $cs(t^\mathcal{F})$ and $prices(t^\mathcal{F})$ as above.

In the running example, $cs(I^\mathcal{F}) = \{(t, l_1)\}$, $prices(I^\mathcal{F})[(p, l_1)] = 0$, $prices(I^\mathcal{F})[(p, t)] = 1$, and $prices(I^\mathcal{F})[(p, l_i)] = \infty$, for all $i \neq 1$. Observe that $prices(I^\mathcal{F})[(p, t)]$ represents the cost of a *possible* package move, not a move we have already committed to. The actions applicable to $I^\mathcal{F}$ are *drive*(l_1, l_i). Applying any such action, in the outcome decoupled state $s^\mathcal{F}$ we have $prices(s^\mathcal{F})[(p, l_i)] = 2$, while all other prices remain the same. If we apply *drive*(l_1, l_2), then $s^\mathcal{F}$ is a goal decoupled state. The global plan is then extracted from $s^\mathcal{F}$ by augmenting the center path $cp(s^\mathcal{F}) = \langle drive(l_1, l_2) \rangle$ with the compliant goal leaf path $\langle load(l_1), unload(l_2) \rangle$.

A *completion plan* for $s^\mathcal{F}$ consists of a center path π^C leading from $s^\mathcal{F}$ to some goal center state, augmented with goal leaf paths compliant with $cp(s^\mathcal{F}) \circ \pi^C$. That is, we collect the postfix path for the center, and the complete path for each leaf. The *completion cost* of $s^\mathcal{F}$, denoted $h^{\mathcal{F}*}(s^\mathcal{F})$, is defined as the cost of a cheapest completion plan for $s^\mathcal{F}$. By $d^{\mathcal{F}*}(s^\mathcal{F})$, we denote the minimum, over all optimal completion plans π^F, of the number of center actions (decoupled-state transitions) in π^F.

Decoupled State Dominance

A binary relation $\preceq$ over decoupled states is a *decoupled dominance relation* if $s^\mathcal{F} \preceq t^\mathcal{F}$ implies that $h^{\mathcal{F}*}(s^\mathcal{F}) \geq h^{\mathcal{F}*}(t^\mathcal{F})$ and $d^{\mathcal{F}*}(s^\mathcal{F}) \geq d^{\mathcal{F}*}(t^\mathcal{F})$. In *dominance pruning*, given such a relation $\preceq$, we prune a state $s^\mathcal{F}$ at generation time if we have already seen another state $t^\mathcal{F}$ (i. e., $t^\mathcal{F}$ is in the open or closed list) such that $s^\mathcal{F} \preceq t^\mathcal{F}$ and $g(s^\mathcal{F}) \geq g(t^\mathcal{F})$. Intuitively, $t^\mathcal{F}$ dominates $s^\mathcal{F}$ if it has an at least equally good completion plan and center path. The center path condition is needed only in the presence of 0-cost actions, and ensures that the completion plan for $t^\mathcal{F}$ does not have to traverse $s^\mathcal{F}$. If $t^\mathcal{F}$ can be reached with equal or better g-cost, pruning $s^\mathcal{F}$ preserves completeness and optimality of the search algorithm.

We derive practical decoupled dominance relations by efficiently testable sufficient criteria. The relations differ in terms of their pruning power. We capture their relative power with two simple terms of two simple notions. First, we say that $\preceq'$ *subsumes* $\preceq$ if $\preceq' \supseteq \preceq$, i. e., if $\preceq'$ recognizes every occurrence of dominance recognized by $\preceq$. Second, we say that $\preceq'$ is *exponentially separated* from $\preceq$ if there exists a family of planning tasks in which the decoupled state space is exponential in the size of the input task under dominance pruning using $\preceq$ and polynomial when using $\preceq'$.[2] We will devise several decoupled dominance relations, weaker and stronger ones. Weaker relations are useful in practice (only) when they cause less computational overhead.

[1] Pricing functions can be maintained in time low-order polynomial in the size of the individual leaf state spaces. See GH for details.

[2] More precisely, as the pruning depends on the expansion order: in which this statement is true for any expansion order.

Previous work only considered what we will refer to as the *basic* decoupled dominance relation, denoted $\preceq_B$.

Definition 1 ($\preceq_B$ relation) *$\preceq_B$ is the relation over decoupled states defined by $s^\mathcal{F} \preceq_B t^\mathcal{F}$ iff $cs(s^\mathcal{F}) = cs(t^\mathcal{F})$ and, for all $s^L \in S^L$, $prices(s^\mathcal{F})[s^L] \geq prices(t^\mathcal{F})[s^L]$.*

This method simply does a point-wise comparison between $prices(s^\mathcal{F})$ and $prices(t^\mathcal{F})$, whenever both have the same center state. Basic dominance pruning often helps to reduce search effort, but is unnecessarily restrictive in its insistence on *all* leaf prices being cheaper. This is inappropriate in cases where $s^\mathcal{F}$ has some irrelevant cheaper prices. It may, indeed, cause exponential blow-ups as, e. g., in our running example.

The standard state space in our running example is small, since $|\mathcal{V}| = 2$. Yet the decoupled state space has size exponential in the number n of locations. Through the leaf state prices, the decoupled states "remember" the locations visited by the truck in the past. For example, the decoupled state reached through the center sequence $\langle drive(l_1, l_3), drive(l_3, l_4)\rangle$ has finite prices for (p, l_1), (p, t), (p, l_3), and (p, l_4), and price ∞ elsewhere; while the decoupled state reached through the sequence $\langle drive(l_1, l_4)\rangle$ has finite prices for (p, l_1), (p, t), and (p, l_4). Intuitively, the difference between the two pricing functions does not matter, because, with initial location l_1 and goal location l_2, the prices for $(p, l_i), i > 2$ are irrelevant. But without recognizing this fact, the decoupled state space enumerates (pricing functions corresponding to) every combination of visited locations.

It is remarkable here that the blow-up occurs in a simple Logistics task. This is a new insight. GH already pointed out the risk of blow-ups, but only in complex artificial examples. On IPC benchmarks, empirically the decoupled state space always is smaller than the standard one. Our insight here is that this is not because blow-ups don't occur, but because the blow-ups (e. g. remembering truck histories) are hidden behind the gains (e. g. not enumerating combinations of package locations). Indeed, in the standard IPC Logistics benchmarks, the blow-up above occurs for all non-airport locations within every city, and these blow-ups multiply across cities. All our advanced dominance pruning methods get rid of this blow-up (though none guarantees to avoid blow-ups in general).

Frontier-Based Dominance

Our first dominance relation is based on the idea that differing prices on a leaf state s^L do not matter if "s^L has no purpose". In our running example, say that we are checking whether $s^\mathcal{F} \preceq t^\mathcal{F}$ and $prices(s^\mathcal{F})[(p, l_3)] = 2$ while $prices(t^\mathcal{F})[(p, l_3)] = \infty$, and thus $s^\mathcal{F} \not\preceq_B t^\mathcal{F}$. However, say that $prices(s^\mathcal{F})[(p, t)] = 1$. Then the cheaper price for (p, l_3) in $s^\mathcal{F}$ does not matter, because the only purpose of having the package at l_3 is to load it into the truck. Indeed, the only outgoing transition of the leaf state (p, l_3) leads to (p, t).

We capture the relevant leaf states in $s^\mathcal{F}$ in terms of its *frontier*: those leaf states that are either themselves relevant (this applies only to the goal leaf state), or that can still contribute to achieving cheaper prices somewhere.

Definition 2 (Frontier) *We define the* frontier *of a decoupled state $s^\mathcal{F}$, $F(s^\mathcal{F}) \subseteq S^L$ as $F(s^\mathcal{F}) := \{s^L_G\} \cup \{s^L \mid \exists s^L \xrightarrow{a} t^L : prices(s^\mathcal{F})[s^L] + \mathsf{cost}(a) < prices(s^\mathcal{F})[t^L]\}$.*

We now obtain a decoupled dominance relation by comparing prices only on the frontier of $s^\mathcal{F}$:

Definition 3 ($\preceq_F$ relation) *$\preceq_F$ is the relation over decoupled states defined by $s^\mathcal{F} \preceq_F t^\mathcal{F}$ iff $cs(s^\mathcal{F}) = cs(t^\mathcal{F})$ and, for all $s^L \in F(s^\mathcal{F})$, $prices(s^\mathcal{F})[s^L] \geq prices(t^\mathcal{F})[s^L]$.*

Theorem 1 *$\preceq_F$ is a decoupled dominance relation.*

Comparing the prices on the frontier is enough because, in any completion plan for $s^\mathcal{F}$, if a compliant leaf path π^L decreases the price of the goal leaf state (e. g., from ∞ to some finite value), then π^L must pass through a frontier state s^L. Hence, in a completion plan for $t^\mathcal{F}$, we can use the postfix behind s^L. This completion plan can only be better than that for $s^\mathcal{F}$ because $prices(s^\mathcal{F})[s^L] \geq prices(t^\mathcal{F})[s^L]$.

It is easy to see that $\preceq_F$ is strictly better than $\preceq_B$:

Theorem 2 *$\preceq_F$ subsumes $\preceq_B$ and is exponentially separated from it.*

The first part of this claim is trivial as both relations are based on comparing prices, but $\preceq_F$ does so on a subset of leaf states. A task family demonstrating the second part of the claim is our running example. The only leaf action applicable in any leaf state (p, l_i) is $load(l_i)$, leading to (p, t). However, for any reachable $s^\mathcal{F}$, we have $prices(s^\mathcal{F})[(p, t)] = 1$ because this price is already achieved in the initial state, and prices can only decrease. So the only possible frontier state, apart from (p, t), is the goal (p, l_2). But only two different prices are reachable for (p, l_2), namely ∞ and 2. This shows the claim.

Effective-Price Dominance

Our next method appears orthogonal to frontier-based dominance at first sight, but turns out to subsume it. The method is based on replacing the prices in $t^\mathcal{F}$, i. e., the dominating state in the comparison $s^\mathcal{F} \preceq t^\mathcal{F}$, with smaller *effective* prices, denoted $Eprices(t^\mathcal{F})$. We then simply compare all such prices:

Definition 4 ($\preceq_E$ relation) *$\preceq_E$ is the relation over decoupled states defined by $s^\mathcal{F} \preceq_E t^\mathcal{F}$ iff $cs(s^\mathcal{F}) = cs(t^\mathcal{F})$ and, for all $s^L \in S^L$, $prices(s^\mathcal{F})[s^L] \geq Eprices(t^\mathcal{F})[s^L]$.*

The modified comparison is sound because the effective prices are designed to preserve $h^{\mathcal{F}*}(t^\mathcal{F})$. Precisely: *(*) For any center path π^C starting in $t^\mathcal{F}$, and for any leaf state s^L of leaf F^L, if π^L_s is a π^C-compliant leaf path from s^L to s^L_G, then there exists a path π^L from s^L_I to s^L_G that complies with $cp(t^\mathcal{F}) \circ \pi^C$ such that*

$\mathsf{cost}(\pi^L) \leq \mathit{Eprices}(t^{\mathcal{F}})[s^L] + \mathsf{cost}(\pi^L_s)$. In other words, if $\mathit{prices}(t^{\mathcal{F}})[s^L] > \mathit{Eprices}(t^{\mathcal{F}})[s^L]$, then any completion plan can be modified to use some other leaf state which does provide a total price of $\mathit{Eprices}(t^{\mathcal{F}})[s^L] + \mathsf{cost}(\pi^L_s)$ or less.

It turns out that this can be ensured with the following simple definition. We define $\mathit{Eprices}(t^{\mathcal{F}})$ as the point-wise minimum pricing function p that satisfies:

$$p[s^L] = \begin{cases} \mathit{prices}(t^{\mathcal{F}})[s^L] & \text{if } s^L = s^L_G \\ \min\{\mathit{prices}(t^{\mathcal{F}})[s^L], & \\ \quad \max_{s^L \xrightarrow{a} t^L} \left(p[t^L] - \mathsf{cost}(a)\right)\} & \text{otherwise} \end{cases}$$

For each F^L, $\mathit{Eprices}(t^{\mathcal{F}})$ can be computed by a simple backwards algorithm starting at the goal leaf state s^L_G. To illustrate the definition, consider any $t^{\mathcal{F}}$ in our running example. The price of (p, t) is 1, and its effective price also is 1 because its successor leaf state $s^L_G = (p, l_2)$ always has effective price ≥ 2. For any irrelevant location l_i, $i > 2$, however, due to the transition to (p, t) whose effective price is 1, we get $\mathit{Eprices}(t^{\mathcal{F}})[(p, l_i)] = 0$ regardless of what the actual price of (p, l_i) in $t^{\mathcal{F}}$ is. The effective price 0 is sound because, in any completion plan for $t^{\mathcal{F}}$ starting with $\mathit{load}(l_i)$, we can use $\mathit{load}(l_1)$ instead to get (p, t) with price 1.

Theorem 3 *$\preceq_E$ is a decoupled dominance relation.*

To prove Theorem 3, observe that, whenever $s^{\mathcal{F}} \preceq_E t^{\mathcal{F}}$, given a completion plan for $s^{\mathcal{F}}$, we can construct an equally good completion plan for $t^{\mathcal{F}}$ by using the same center path π^C, and, with (*) above, constructing equally good or cheaper compliant goal leaf paths. It remains to prove (*). Consider any $t^{\mathcal{F}}$, center path π^C, leaf state s^L, and π^C-compliant goal leaf path π^L_s starting in s^L. In our example, e. g., say $t^{\mathcal{F}}$ is reached from $I^{\mathcal{F}}$ by applying $\mathit{drive}(l_1, l_3)$; that $\pi^C = \langle \mathit{drive}(l_3, l_2) \rangle$; that $s^L = (p, l_3)$; and that $\pi^L_s = \langle \mathit{load}(l_3), \mathit{unload}(l_2) \rangle$. Then, exists $\pi^L = \langle \mathit{load}(l_1), \mathit{unload}(l_2) \rangle$ that is compliant with $\mathit{cp}(t^{\mathcal{F}}) \circ \pi^C$.

Formally, denote $\pi^L_s = \langle a_1, \dots, a_n \rangle$ and denote the leaf states it traverses by $s^L = s^L_0, \dots, s^L_n = s^L_G$. Observe that, as $\mathit{Eprices}(t^{\mathcal{F}})[s^L_n] = \mathit{prices}(t^{\mathcal{F}})[s^L_n]$, π^L_s necessarily passes through a leaf state s^L_i whose effective and actual prices in $t^{\mathcal{F}}$ are identical. Let i be the smallest index for which that is so. Then, for all $j < i$, $\mathit{Eprices}(t^{\mathcal{F}})[s^L_j] \neq \mathit{prices}(t^{\mathcal{F}})[s^L_j]$, and thus by the definition of effective prices we have that $\mathit{Eprices}(t^{\mathcal{F}})[s^L_j] \geq \mathit{Eprices}(t^{\mathcal{F}})[s^L_{j+1}] - \mathsf{cost}(a_{j+1})$. Accumulating these inequalities, we get *(**)* $\mathit{Eprices}(t^{\mathcal{F}})[s^L_0] \geq \mathit{Eprices}(t^{\mathcal{F}})[s^L_i] - \sum_{j=1}^{i} \mathsf{cost}(a_j)$. Consider now the path π^L from s^L_I to s^L_G constructed as the concatenation of: a cheapest $\mathit{cp}(t^{\mathcal{F}})$-compliant path to s^L_i (in our example, $\langle \mathit{load}(l_1) \rangle$); with the postfix of π^L_s behind s^L_i (in our example, $\langle \mathit{unload}(l_2) \rangle$). Then $\mathsf{cost}(\pi^L) = \mathit{prices}(t^{\mathcal{F}})[s^L_i] + \sum_{j=i+1}^{n} \mathsf{cost}(a_j)$. As $\mathit{Eprices}(t^{\mathcal{F}})[s^L_i] = \mathit{prices}(t^{\mathcal{F}})[s^L_i]$, we get $\mathsf{cost}(\pi^L) = \mathit{Eprices}(t^{\mathcal{F}})[s^L_i] + \sum_{j=i+1}^{n} \mathsf{cost}(a_j)$. With (**), we get the desired property that $\mathsf{cost}(\pi^L) \leq \mathit{Eprices}(t^{\mathcal{F}})[s^L_0] + \sum_{j=1}^{i} \mathsf{cost}(a_j) + \sum_{j=i+1}^{n} \mathsf{cost}(a_j) = \mathit{Eprices}(t^{\mathcal{F}})[s^L] + \mathsf{cost}(\pi^L_s)$, concluding the proof.

Theorem 4 *$\preceq_E$ subsumes $\preceq_F$ and is exponentially separated from it.*

To prove the exponential separation, we extend our running example with a *teleport*(l_i, l_j) action, for $i, j > 2$, that moves the package between irrelevant locations if the truck is at l_2. Then, as long as l_2 and at least one such l_i have not been visited yet, all leaf states (p, l_i) for $i > 2$ with finite price are in the frontier, and $\preceq_F$ suffers from the same blow-up as $\preceq_B$. The effective prices of (p, l_i), however, remain 0 as before.

To see that $\preceq_E$ subsumes $\preceq_F$, observe that the former can be viewed as a recursive version of the latter, when reformulating the frontier condition to "$\exists s^L \xrightarrow{a} t^L : p[s^L] < p[t^L] - \mathsf{cost}(a)$". Formally, one can show that, if $\mathit{Eprices}(t^{\mathcal{F}})[s^L] \leq \mathit{prices}(s^{\mathcal{F}})[s^L]$ holds for all frontier states $s^L \in F(s^{\mathcal{F}})$, then it also holds for all non-frontier states $s^L \notin F(s^{\mathcal{F}})$. This shows the claim as, for $s^{\mathcal{F}} \preceq_F t^{\mathcal{F}}$, we have $\mathit{prices}(s^{\mathcal{F}})[s^L] \geq \mathit{prices}(t^{\mathcal{F}})[s^L]$ on $s^L \in F(s^{\mathcal{F}})$, and thus $\mathit{prices}(s^{\mathcal{F}})[s^L] \geq \mathit{Eprices}(t^{\mathcal{F}})[s^L]$ on these states.

Note that, with the above, to evaluate $\preceq_E$ it suffices to compare the price of $s^{\mathcal{F}}$ vs. effective price of $t^{\mathcal{F}}$ on $F(s^{\mathcal{F}})$. This is equivalent to, but faster than, comparing all prices.

Simulation-Based Dominance

We use the concept of simulation relations (Milner 1971; Gentilini *et al.* 2003) *on leaf state spaces* in order to identify leaf states t^L which can do everything that another leaf state s^L can do.[3] In this situation, suppose that we are checking whether $s^{\mathcal{F}} \preceq t^{\mathcal{F}}$, and $\mathit{prices}(t^{\mathcal{F}})[s^L] > \mathit{prices}(s^{\mathcal{F}})[s^L]$, but $\mathit{prices}(t^{\mathcal{F}})[t^L] \leq \mathit{prices}(s^{\mathcal{F}})[s^L]$. Then $t^{\mathcal{F}}$ can still dominate $s^{\mathcal{F}}$, because if a solution for $s^{\mathcal{F}}$ relies on s^L, then starting from $t^{\mathcal{F}}$ we can use t^L instead.

Definition 5 (Leaf simulation) *Let F^L be a leaf factor. A binary relation $\preceq^L$ on F^L leaf states is a* leaf simulation *if: $s^L_G \not\preceq^L s^L$ for all $s^L \neq s^L_G$; and whenever $s^L_1 \preceq^L t^L_1$, for every transition $s^L_1 \xrightarrow{a} s^L_2$ either (i) $s^L_2 \preceq^L t^L_1$ or (ii) there exists a transition $t^L_1 \xrightarrow{a'} t^L_2$ s.t. $s^L_2 \preceq^L t^L_2$, $\mathsf{pre}[F^C](a') \subseteq \mathsf{pre}[F^C](a)$, and $\mathsf{cost}(a') \leq \mathsf{cost}(a)$.*

This follows common notions, except for (i) which, intuitively, "allows t^L_1 to stay where it is", and except for allowing in (ii) different actions a' so long as they are at least as good in terms of center precondition and cost.

It is easy to see that, whenever $s^L \preceq^L t^L$, if a leaf path π^L_s starting in s^L complies with a center path π^C, then there exists a π^C-compliant leaf path π^L_t starting in t^L s.t. $\mathsf{cost}(\pi^L_t) \leq \mathsf{cost}(\pi^L_s)$. Consequently, we allow s^L to take a cheaper price from any leaf state that simulates it:

[3]This is inspired by, but differs in scope and purpose from, the use of simulation relations on the state space for dominance pruning in standard search (Torralba and Hoffmann 2015).

Definition 6 (**$\preceq_S$ Relation**) *The relation $\preceq_S$ over decoupled states is defined by $s^{\mathcal{F}} \preceq_S t^{\mathcal{F}}$ iff $cs(s^{\mathcal{F}}) = cs(t^{\mathcal{F}})$ and, for all $s^L \in S^L$, $prices(s^{\mathcal{F}})[s^L] \geq \min_{s^L \preceq^L t^L} prices(t^{\mathcal{F}})[t^L]$.*

Theorem 5 *$\preceq_S$ is a decoupled dominance relation.*

It is easy to see that this is strictly better than $\preceq_B$:

Theorem 6 *$\preceq_S$ subsumes $\preceq_B$ and is exponentially separated from it.*

The first part of this claim holds simply because $\preceq^L$ is reflexive (and therefore $\min_{s^L \preceq^L t^L} prices(t^{\mathcal{F}})[t^L] \leq prices(t^{\mathcal{F}})[s^L]$). For the second part, we use again our running example. Leaf simulation captures that $(p, l_i) \preceq^L (p, t)$ for all $i > 2$, since (p, t) is the only successor of any (p, l_i) and naturally $(p, t) \preceq^L (p, t)$. So, $\preceq_S$ reduces the price of such (p, l_i) to 1, avoiding the exponential blow-up.

Inspired by (Torralba and Kissmann 2015), we also employ leaf simulation to remove superfluous leaf states and leaf actions, discovering transitions that can be replaced by other transitions, then running a reachability check on the leaf state space (details are in the TR). This reduces leaf state space size, and may sometimes improve the heuristic function due to the removal of some actions.

Method Interrelations and Combination

We have already established the relation of our methods relative to $\preceq_B$, as well as the relation between $\preceq_E$ and $\preceq_F$. We next design a combination $\preceq_{ES}$ of $\preceq_E$ and $\preceq_S$, with their respective strengths, and we establish the remaining method interrelations. Figure 1 provides the overall picture.

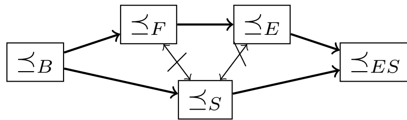


Figure 1: Summary of method interrelations. "$A \rightarrow B$": B subsumes A and is exponentially separated from it. "$A \not\leftrightarrow B$": A is exponentially separated from B and vice versa.

The combined relation $\preceq_{ES}$ is obtained by modifying the effective prices underlying $\preceq_E$, enriching their definition with a leaf simulation, $\preceq^L$. We define $ESprices(t^{\mathcal{F}})$ as the point-wise minimum pricing function p that satisfies:

$$p[s^L] = \begin{cases} prices(t^{\mathcal{F}})[s^L] & \text{if } s^L = s^L_G \\ \min\{\min_{s^L \preceq^L t^L} prices(t^{\mathcal{F}})[t^L], \\ \quad \max_{s^L \xrightarrow{a} t^L} \big(p[t^L] - \mathsf{cost}(a)\big)\} & \text{otherwise} \end{cases}$$

We integrate the information from a leaf simulation into the effective prices by allowing s^L to take cheaper prices from simulating states t^L. This amounts to substituting $prices(t^{\mathcal{F}})[s^L]$ with $\min_{s^L \preceq^L t^L} prices(t^{\mathcal{F}})[t^L]$ in the equation. We thus obtain, again, a decoupled dominance relation:

Definition 7 (**$\preceq_{ES}$ Relation**) *$\preceq_{ES}$ is the relation over decoupled states defined by $s^{\mathcal{F}} \preceq_{ES} t^{\mathcal{F}}$ iff $cs(s^{\mathcal{F}}) = cs(t^{\mathcal{F}})$ and, for all $s^L \in S^L$, $prices(s^{\mathcal{F}})[s^L] \geq ESprices(t^{\mathcal{F}})[s^L]$.*

Theorem 7 *$\preceq_{ES}$ is a decoupled dominance relation.*

Theorem 7 is shown by adapting the property (*) underlying the proof of Theorem 3. Say $\pi^L_s = \langle a_1, \dots, a_n \rangle$ is a π^C-compliant goal leaf path starting in s^L, traversing the leaf states $s^L = s^L_0, \dots, s^L_n = s^L_G$. Then, with the same arguments as before, there exists i such that (a) $ESprices(t^{\mathcal{F}})[s^L_0] \geq ESprices(t^{\mathcal{F}})[s^L_i] - \sum_{j=1}^{i} \mathsf{cost}(a_i)$, and (b) $ESprices(t^{\mathcal{F}})[s^L_i] = \min_{s^L_i \preceq^L t^L} prices(t^{\mathcal{F}})[t^L]$. We construct our desired path π^L from s^L_I to s^L_G by a cheapest $cp(t^{\mathcal{F}})$-compliant path to a t^L minimizing the expression in (b), concatenated with a π^C-compliant goal leaf path π^L_t starting in t^L where $\mathsf{cost}(\pi^L_t) \leq \mathsf{cost}(\pi^L_s)$. Such π^L_t exists by the properties of leaf simulation, as in Theorem 5.

$\preceq_{ES}$ subsumes each of its components. The exponential separations therefore follow directly from the individual ones:

Theorem 8 *$\preceq_{ES}$ subsumes $\preceq_E$ and $\preceq_S$, and is exponentially separated from each of them.*

One can also construct cases where $\preceq_{ES}$ yields an exponentially stronger reduction than *both* $\preceq_E$ and $\preceq_S$, i. e., where $\preceq_{ES}$ is strictly more than the sum of its components. We complete our analysis by filling in the missing cases:

Theorem 9 *$\preceq_S$ is exponentially separated from $\preceq_E$, and therefore also from $\preceq_F$. $\preceq_F$, and therefore also $\preceq_E$, is exponentially separated from $\preceq_S$.*

Experiments

We implemented our dominance pruning methods within the fork-decoupled search variant of FD (Helmert 2006) by GH. Our baseline is GH's basic pruning $\preceq_B$. For simplicity, we stick to the factoring strategy used by GH. This method greedily computes a factoring that maximizes the number of leaf factors. In case there are less than two leaves, the method *abstains* from solving a task. The rationale behind this is that the main advantage of decoupled search originates from not having to enumerate leaf state combinations across *multiple* leaf factors. Like GH, we show results on all IPC domains up to and including 2014 where the strategy does not abstain.

We focus on optimal planning, the main purpose of optimality-preserving pruning. We run a blind heuristic to identify the influence of different pruning methods per se, and we run LM-cut (Helmert and Domshlak 2009) as a state-of-the-art heuristic. GH introduced two decoupled variants of A*, "Fork-Decoupled" A* and "Anytime Fork-Root" A*, which to simplify terminology we will refer to as *Decoupled A* (DA*)* and *Anytime Decoupled A* (ADA*)*. DA* is a direct application of A* to the decoupled state space. ADA* orders the open list based on the heuristic estimate of

remaining center-cost, uses the heuristic estimate of remaining global-cost for pruning against the best solution so far, and runs until the open list is empty. Both algorithms result in similar coverage, with moderate differences in some domains. Our techniques turn out to be more beneficial for ADA*, which tends to have larger search spaces but less per-node runtime than DA*. We show detailed data for ADA*, and include data for baseline DA* (with $\preceq_B$) for comparison. All experiments are run on a cluster of Intel E5-2660 machines running at 2.20 GHz, with time (memory) cut-offs of 30 minutes (4 GB).

		Blind Heuristic					LM-cut					
		ADA*					DA*	ADA*				
Domain	#	$\preceq_B$	$\preceq_F$	$\preceq_E$	$\preceq_S$	$\preceq_{ES}$	$\preceq_B$	$\preceq_B$	$\preceq_F$	$\preceq_E$	$\preceq_S$	$\preceq_{ES}$
Driverlog	20	11	11	11	11	11	13	13	13	13	13	13
Logistics00	28	22	22	22	22	22	**28**	25	25	27	26	**28**
Logistics98	35	4	4	**5**	**5**	**5**	6	6	6	6	6	6
Miconic	145	36	**45**	**45**	**45**	**45**	135	135	135	135	135	135
NoMystery	20	17	**20**	**20**	**20**	**20**	20	20	20	20	20	20
Pathways	29	3	3	3	3	3	4	4	4	4	4	4
Rovers	40	**7**	6	6	**7**	6	9	9	9	9	9	9
Satellite	36	**6**	**6**	**6**	**6**	5	7	**9**	**9**	8	**9**	**9**
TPP	27	**23**	**23**	22	**23**	22	18	**23**	**23**	22	22	22
Woodwork08	13	5	5	5	5	5	10	**11**	**11**	**11**	**11**	**11**
Woodwork11	5	1	1	1	1	1	4	**5**	**5**	**5**	**5**	**5**
Zenotravel	20	11	11	**12**	**12**	**12**	**13**	11	11	12	12	**13**
$\sum$	418	146	157	158	**160**	157	267	271	271	272	272	**275**

Table 1: Coverage data.

Table 1 shows the number of instances solved, comparing to both baselines DA* and ADA*. Data for DA* with the blind heuristic is not shown as it is identical to that for ADA*. The main gain for blind search stems from Miconic (+9), and NoMystery (+3). When using LM-cut, the advantage over $\preceq_B$ is much smaller. We still gain +3 (+2) instances in Logistics00 (Zenotravel). In Satellite and TPP, we lose 1 instance in some configurations due to overhead at no search space reduction. $\preceq_{ES}$ reliably removes the disadvantages of ADA* relative to DA*, and is best in the overall. We never strictly improve coverage over both baselines, though. As we shall see below, this is due to benchmark scaling, i. e., there *are* domains where runtime is improved over both baselines.

We next analyze the search space size reduction (top part of Table 2). In general, the blind heuristic has more margin of improvement except in Logistics98, where the improvement with LM-cut gets magnified due to the relevance analysis performed when enabling $\preceq_S$. In that domain, removing irrelevant leaf states and leaf actions renders LM-cut a lot stronger.[4] Regarding the relative behavior of pruning techniques, in two domains, namely Miconic and NoMystery, already the simplest technique ($\preceq_F$) gets the maximal improvement factor. In four domains, enabling effective-price pruning on top of frontier pruning results in additional pruning. Combining all techniques in $\preceq_{ES}$ always inherits the strongest search space reduction of its components and in Logistics with LM-cut, it often is strictly better.

Consider now runtime, Table 2 bottom. One key observation is that, whenever the search space is reduced, the same holds for runtime, even for small search space reduction factors like, e. g., in Zenotravel. Remarkably, in some domains (e. g. Woodworking) where no search reduction is obtained, runtime decreases nevertheless for some simple methods such as $\preceq_F$. This is due to the cheaper dominance check – prices are compared only on *frontier* leaf states. There are also some bad cases, though, mainly in TPP, but also in Pathways, Rovers, and Satellite. These are also the domains in which coverage slightly decreases. What makes these domains special is the structure of their leaf state spaces. In Pathways, Rovers, and Satellite, all leaves are single variables with a single transition, $s_I^L \rightarrow s_G^L$, so there is no room for improvement. In TPP, the leaf state spaces are quite large (up to 5000 states), so our methods incur substantial overhead, but are unable to perform pruning. Presumably, this is because most of the leaf states can play a role in optimally reaching the goal.

Coming back to our previous observation that coverage is never improved over both baselines, the runtime analysis reveals an improvement over both baselines in several domains. ADA* with $\preceq_S$ is faster than DA* with $\preceq_B$ in all domains except Zenotravel, where the geomean per-instance runtime factor is 0.7. The other factors are: Driverlog 2.3; Logistics00 2.3; Logistics98 3.4; Miconic 2.7; NoMystery 3.2; Pathways 1.1; Rovers 2.1; Satellite 2.9; TPP 23.2; Woodworking08 1.4; and Woodworking11 2.0. In particular, in Driverlog, both Logistics domains, NoMystery, and Woodworking11, ADA* with $\preceq_S$ improves runtime over both baselines.

Finally, consider the use of our pruning methods in DA*. For blind search, the numbers are almost identical to those for ADA* in Table 2, as DA* and ADA* differ mainly in their use of a (non-trivial) heuristic. With LM-cut, the pruning methods do not work as well for DA*. For example, for $\preceq_S$, the geomean per-instance runtime factors are: Driverlog 1.8; Logistics00 and Logistics98 2.5; NoMystery 2.0; TPP 0.9; Woodworking08 0.9; Woodworking11 1.3; Zenotravel 1.2; and 1.0 in the other domains. The picture is similar for the other pruning methods. The big runtime advantages observed with ADA* vanish, but the method also becomes less risky, i. e., the big runtime disadvantage in TPP vanishes as well. This makes sense since DA* searches less nodes (it has less potential for pruning) while spending more time on each node (making the dominance-checking overhead less pronounced).

Conclusion

Dominance pruning methods can be quite useful for decoupled search. Our analysis of such methods is fairly complete, although of course other variants may be thinkable. More pressingly, the question remains whether there exist duplicate checking methods guaranteeing to avoid all blow-ups.

[4]It may be surprising that, elsewhere, the improvements in Logistics are moderate, despite the inherent blow-up we explained earlier. This is because, in the commonly solved instances, the number of non-airport locations in each city is very small, mostly 1.

	Expansions with Blind Heuristic: Improvement factor relative to $\preceq_B$													Expansions with LM-cut: Improvement factor relative to $\preceq_B$												
		$\preceq_F$			$\preceq_E$			$\preceq_S$			$\preceq_{ES}$				$\preceq_F$			$\preceq_E$			$\preceq_S$			$\preceq_{ES}$		
Domain	#	$\sum$D	GM	max	$\sum$D	GM	max	$\sum$D	GM	max	$\sum$D	GM	max	#	$\sum$D	GM	max	$\sum$D	GM	max	$\sum$D	GM	max	$\sum$D	GM	max
Driverlog	11	1.0	1.0	1.0	5.0	1.8	6.5	2.4	1.3	2.8	5.0	1.8	6.5	13	1.0	1.0	1.0	2.4	1.3	4.3	1.9	1.2	3.4	2.4	1.3	4.3
Logistics00	22	1.2	1.0	1.2	2.5	1.4	3.8	2.5	1.4	3.8	2.5	1.4	3.8	25	1.0	1.0	1.0	2.1	1.2	2.3	1.4	1.3	3.0	2.2	1.4	3.0
Logistics98	4	1.0	1.0	1.0	3.9	2.1	4.2	2.3	1.7	2.4	3.9	2.1	4.2	6	1.0	1.0	1.0	1.7	1.3	1.7	109.8	10.2	1245.2	134.7	10.8	1245.2
Miconic	36	3.3	1.7	5.2	3.3	1.7	5.2	3.3	1.7	5.2	3.3	1.7	5.2	135	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
NoMystery	17	4.4	1.7	8.5	4.4	1.7	8.5	4.4	1.7	8.5	4.4	1.7	8.5	20	6.3	1.7	9.2	6.3	1.7	9.2	6.8	1.9	9.3	6.8	1.9	9.3
TPP	22	1.0	1.0	1.0	1.0	1.0	1.2	1.0	1.0	1.0	1.0	1.0	1.2	22	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Zenotravel	11	1.0	1.0	1.0	1.4	1.1	1.6	1.3	1.1	1.5	1.4	1.1	1.6	11	1.0	1.0	1.0	1.2	1.1	1.4	1.2	1.0	1.3	1.2	1.1	1.4

	Runtime with Blind Heuristic: Improvement factor relative to $\preceq_B$													Runtime with LM-cut: Improvement factor relative to $\preceq_B$												
		$\preceq_F$			$\preceq_E$			$\preceq_S$			$\preceq_{ES}$				$\preceq_F$			$\preceq_E$			$\preceq_S$			$\preceq_{ES}$		
Domain	#	$\sum$D	GM	max	$\sum$D	GM	max	$\sum$D	GM	max	$\sum$D	GM	max	#	$\sum$D	GM	max	$\sum$D	GM	max	$\sum$D	GM	max	$\sum$D	GM	max
Driverlog	9	0.9	0.9	1.0	30.7	2.6	38.9	10.3	2.2	14.4	35.3	2.9	47.5	5	0.8	0.9	1.0	5.5	2.6	14.3	4.4	2.5	11.3	5.5	2.7	14.6
Logistics00	7	1.4	1.3	1.5	6.4	5.9	15.2	8.4	8.3	22.5	7.5	7.0	19.7	9	0.9	0.9	0.9	3.8	1.5	4.6	2.7	3.7	6.4	4.1	3.5	5.0
Logistics98	3	0.8	0.8	0.8	21.2	4.1	22.4	12.1	5.4	12.3	26.4	6.2	27.5	4	0.9	0.9	0.9	2.2	1.2	2.2	895.9	30.4	2643.9	750.2	26.2	2259.3
Miconic	19	24.0	10.0	53.9	24.3	9.0	47.9	22.6	8.6	45.7	23.5	8.8	47.0	81	0.9	1.0	1.2	1.0	0.9	1.1	1.0	1.0	1.2	0.9	0.9	1.0
NoMystery	9	47.3	5.6	157.1	36.2	4.1	118.8	64.2	7.4	210.2	53.7	6.0	182.7	12	13.3	3.0	21.0	12.6	2.9	22.4	16.2	3.8	28.9	14.6	3.6	26.0
Pathways	2	0.9	0.9	0.9	0.7	0.7	0.7	1.0	1.0	1.0	0.6	0.6	0.6	1	0.9	0.9	0.9	0.9	0.9	0.9	1.0	1.0	1.0	0.9	0.9	0.9
Rovers	2	0.8	0.8	0.8	0.5	0.5	0.6	1.0	1.0	1.0	0.5	0.5	0.5	5	0.9	0.9	0.9	0.7	0.7	0.8	1.0	1.0	1.0	0.7	0.7	0.8
Satellite	3	0.9	0.9	1.0	0.6	0.7	0.9	1.0	1.0	1.0	0.5	0.6	0.8	4	1.0	1.0	1.0	0.9	0.8	0.9	1.0	1.0	1.0	0.9	0.8	0.9
TPP	13	0.8	0.8	1.0	0.0	0.1	0.3	0.1	0.3	0.8	0.0	0.1	0.3	11	0.8	0.8	1.0	0.1	0.2	0.4	0.1	0.4	0.8	0.1	0.1	0.3
Woodwork08	2	1.5	1.2	1.5	0.7	0.8	1.0	1.5	0.3	1.5	1.0	0.3	1.0	8	1.0	1.0	1.1	1.0	1.0	1.0	1.2	0.9	1.7	1.1	0.8	1.4
Woodwork11	1	1.5	1.5	1.5	0.7	0.7	0.7	1.5	1.5	1.5	1.0	1.0	1.0	5	1.0	1.0	1.0	1.0	1.0	1.0	1.3	1.2	1.3	1.3	1.2	1.3
Zenotravel	4	0.8	0.8	1.0	1.2	1.2	1.4	1.7	1.8	2.9	1.3	1.3	1.8	4	0.9	0.9	1.0	1.1	1.0	1.2	1.3	1.3	1.6	1.1	1.1	1.3

Table 2: Improvement factor on commonly solved instances relative to $\preceq_B$, using ADA*. We show expansions up to last f-layer (top), and runtime (bottom), with the blind heuristic (left) and LM-cut (right). In the top part, some domains are skipped as all their factors are rounded to 1.0. In the bottom part, we only take into account the instances that are not trivially solved by all planners ($< 0.1s$). $\sum$D: Ratio over the per-domain sum. GM (max): geometric mean (maximum) of per-instance ratios.

Acknowledgments

This work was partially supported by the German Research Foundation (DFG), under grant HO 2169/6-1, "Star-Topology Decoupled State Space Search".

References

Eyal Amir and Barbara Engelhardt. Factored planning. In G. Gottlob, editor, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 929–935, Acapulco, Mexico, August 2003. Morgan Kaufmann.

Christer Bäckström and Bernhard Nebel. Complexity results for SAS$^+$ planning. *Computational Intelligence*, 11(4):625–655, 1995.

Ronen Brafman and Carmel Domshlak. Structure and complexity in planning with unary operators. *Journal of Artificial Intelligence Research*, 18:315–349, 2003.

Ronen Brafman and Carmel Domshlak. On the complexity of planning for agent teams and its implications for single agent planning. *Artificial Intelligence*, 198:52–71, 2013.

Eric Fabre, Loïg Jezequel, Patrik Haslum, and Sylvie Thiébaux. Cost-optimal factored planning: Promises and pitfalls. In Ronen I. Brafman, Hector Geffner, Jörg Hoffmann, and Henry A. Kautz, editors, *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS'10)*, pages 65–72. AAAI Press, 2010.

Raffaella Gentilini, Carla Piazza, and Alberto Policriti. From bisimulation to simulation: Coarsest partition problems. *Journal of Automated Reasoning*, 31(1):73–103, 2003.

Daniel Gnad and Jörg Hoffmann. Beating LM-cut with h^{max} (sometimes): Fork-decoupled state space search. In Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein, editors, *Proceedings of the 25th International Conference on Automated Planning and Scheduling (ICAPS'15)*. AAAI Press, 2015.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, pages 162–169. AAAI Press, 2009.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Peter Jonsson and Christer Bäckström. Incremental planning. In *European Workshop on Planning*, 1995.

Michael Katz and Carmel Domshlak. Structural patterns heuristics via fork decomposition. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric Hansen, editors, *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS'08)*, pages 182–189. AAAI Press, 2008.

Michael Katz and Emil Keyder. Structural patterns beyond forks: Extending the complexity boundaries of clas-

sical planning. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI'12)*, pages 1779–1785, Toronto, ON, Canada, July 2012. AAAI Press.

Elena Kelareva, Olivier Buffet, Jinbo Huang, and Sylvie Thiébaux. Factored planning using decomposition trees. In M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 1942–1947, Hyderabad, India, January 2007. Morgan Kaufmann.

Craig Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

Robin Milner. An algebraic definition of simulation between programs. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (IJCAI'71)*, pages 481–489, London, UK, September 1971. William Kaufmann.

Álvaro Torralba and Jörg Hoffmann. Simulation-based admissible dominance pruning. In Qiang Yang, editor, *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 1689–1695. AAAI Press/IJCAI, 2015.

Álvaro Torralba and Peter Kissmann. Focusing on what really matters: Irrelevance pruning in merge-and-shrink. In Levi Lelis and Roni Stern, editors, *Proceedings of the 8th Annual Symposium on Combinatorial Search (SOCS'15)*, pages 122–130. AAAI Press, 2015.

Álvaro Torralba, Daniel Gnad, Patrick Dubbert, and Jörg Hoffmann. On state-dominance criteria in fork-decoupled search (technical report). Technical report, Saarland University, 2016. Available at `http://fai.cs.uni-saarland.de/hoffmann/papers/ijcai16b-tr.pdf`.

Decoupled Strong Stubborn Sets

Daniel Gnad
Saarland University
Saarbrücken, Germany
gnad@cs.uni-saarland.de

Martin Wehrle
University of Basel
Basel, Switzerland
martin.wehrle@unibas.ch

Jörg Hoffmann
Saarland University
Saarbrücken, Germany
hoffmann@cs.uni-saarland.de

Abstract

Recent work has introduced *fork-decoupled search*, addressing classical planning problems where a single *center* component provides preconditions for several *leaf* components. Given a fixed center path π^C, the leaf moves *compliant* with π^C can then be scheduled independently for each leaf. Fork-decoupled search thus searches over center paths only, maintaining the compliant paths for each leaf separately. This can yield dramatic benefits. It is empirically complementary to partial order reduction via *strong stubborn sets*, in that each method yields its strongest reductions in different benchmarks. Here we show that the two methods can be combined, in the form of strong stubborn sets for fork-decoupled search. This can yield exponential advantages relative to both methods. Empirically, the combination reliably inherits the best of its components, and often outperforms both.

Introduction

In classical AI planning, the task is to find a sequence of actions leading from a given initial state to a state that satisfies a given goal condition, in a large deterministic transition system (the task's *state space*). Gnad and Hoffmann (2015a) (henceforth: GH) have recently introduced a new approach, *fork-decoupled search*, to decompose the state space. The approach relates to *factored planning* (e.g. (Amir and Engelhardt 2003; Kelareva *et al.* 2007; Fabre *et al.* 2010; Brafman and Domshlak 2013)), where the *factors* are disjoint subsets of state variables. Fork-decoupled search assumes that the factors induce a *fork* structure: a single *center* factor provides preconditions for several *leaf* factors, and no other cross-factor interactions exist. As GH show, such a *fork factoring*, if one exists, can be easily identified in a pre-process to planning, based on the task's *causal graph* (e.g. (Knoblock 1994; Jonsson and Bäckström 1995; Brafman and Domshlak 2003; Helmert 2006)).

In a fork factoring, the leaves are "conditionally independent", in the sense that, given a fixed center path π^C, the *compliant* leaf moves – those leaf moves enabled by the preconditions supplied along π^C – can be scheduled independently for each leaf. This can be exploited by searching only over center paths, and maintaining the possible compliant paths separately for each leaf, thus avoiding the enumeration of state combinations across leaves. GH show how to employ standard heuristic search planning algorithms, preserving optimality guarantees. They obtain dramatic benefits in several International Planning Competition (IPC) benchmarks.

Fork-decoupling can be thought of as a reformulation of the search space. *Can known search reduction methods be applied on the reformulated search space as well?* We herein answer this in the affirmative for a prominent reduction method, namely state-of-the-art partial order reduction via *strong stubborn sets (SSS)* (Valmari 1989; Alkhazraji *et al.* 2012; Wehrle and Helmert 2012; 2014). This method prunes applicable actions on states s during (standard, non-decoupled) search, namely those not contained in an SSS for s. The SSS is guaranteed to contain at least one action starting an optimal plan for s, so optimality is preserved.

Fork-decoupled search and SSS yield their respective best reductions in different IPC domains. We show that the two methods are indeed *exponentially separated*, i.e., that there are cases where one yields exponentially stronger reductions than the other. We show how to combine them in the form of *decoupled strong stubborn sets (DSSS)*, for fork-decoupled search. We show that this combination is exponentially separated from both its components. There are cases – not complex artificial examples, but simple variants of the Logistics benchmark – where DSSS yield exponentially stronger reductions than *both* fork-decoupling and SSS. Empirically, DSSS reliably inherit the strengths of each component, and sometimes outperform both. In some cases, DSSS even is "more than the sum of its components", yielding a stronger reduction in fork-decoupled search than SSS do in standard search.

For space reasons, we give proof sketches only. The full proofs are available in an online TR (Gnad *et al.* 2016).

Background

We employ a finite-domain state variable formalization of planning (e.g. (Bäckström and Nebel 1995; Helmert 2006)). A *finite-domain representation* planning task, short FDR task, is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, s_0, s_\star \rangle$. $\mathcal{V}$ is a set of *state variables*, each $v \in \mathcal{V}$ associated with a finite domain $\mathcal{D}(v)$. We identify (partial) variable assignments with sets of variable/value pairs. A complete assignment to $\mathcal{V}$ is a *state*. s_0 is the *initial state*, and the *goal* $s_\star$ is a partial assignment to V. $\mathcal{A}$ is a finite set of *actions*. Each action $a \in \mathcal{A}$ is a triple $\langle pre(a), eff(a), cost(a) \rangle$ where the *precondition* $pre(a)$ and

effect $\mathit{eff}(a)$ are partial assignments to $\mathcal{V}$, with $\mathit{eff}(a) \neq \emptyset$; $cost(a) \in \mathbb{R}^{0+}$ is a's non-negative *cost*.

Given a partial assignment p, by $vars(p) \subseteq \mathcal{V}$ we denote the subset of state variables on which p is defined. For $V \subseteq vars(p)$, by $p[V]$ we denote the assignment to V made by p. Action a is *applicable* in state s if $s \models pre(a)$, i.e., $pre(a) \subseteq s$. Applying a in s changes the value of $v \in vars(\mathit{eff}(a))$ to $\mathit{eff}(a)[v]$, and leaves s unchanged elsewhere. A *plan* for Π is an action sequence π applicable in s_0 and ending in a state s such that $s \models s_\star$. The plan is *optimal* if its summed-up cost, denoted $cost(\pi)$, is minimal among all plans for Π.

We next give a summary of fork-decoupled search. We will often write "decoupled" instead of "fork-decoupled".

A *factoring* $\mathcal{F}$ is a partition of $\mathcal{V}$. $\mathcal{F}$ is a *fork factoring* if $|\mathcal{F}| \geq 2$ and there exists $F^C \in \mathcal{F}$ s.t. the arcs in $\mathcal{F}$'s *interaction graph* are exactly $\{(F^C, F^L) \mid F^L \in \mathcal{F} \setminus \{F^C\}\}$. Here, the interaction graph is the quotient of the task's *causal graph* over $\mathcal{F}$, i.e., it contains an arc (F, F') if there exists $a \in \mathcal{A}$ s.t. $F \cap [vars(pre(a)) \cup vars(\mathit{eff}(a))] \neq \emptyset$ and $F' \cap vars(\mathit{eff}(a)) \neq \emptyset$. We refer to F^C as the *center* of $\mathcal{F}$, and to all other factors $F^L \in \mathcal{F}^L := \mathcal{F} \setminus \{F^C\}$ as its *leaves*.

As a running example, consider a Logistics-style planning task with 1 truck variable t, N package variables p_i, and two locations A and B. The truck and all packages are initially at A, and the goal is for the packages to be at B. The actions (unit costs) are *drive*, *load*, and *unload*, with the usual preconditions and effects (e.g. *load*(t, p_i, A) requires both t and p_i to be at A, and moves p_i into the truck). Setting $\{t\}$ as the center and each $\{p_i\}$ as a leaf, we obtain a fork factoring.

Not every task Π has a fork factoring. We assume GH's approach of analyzing Π's causal graph in a pre-process, identifying a fork factoring if one exists, else *abstaining* from solving Π. In what follows, assume a fork factoring $\mathcal{F}$.

Given the structure of the interaction graph, every action affects (touches in its effect) either only F^C, or only one leaf F^L. We refer to the former kind as *center actions*, and to the latter kind as *leaf actions*. Observe that center actions do not have any preconditions on leaves. Furthermore, if leaf action a affects leaf F^L, then it can be preconditioned only on F^C and F^L, i.e., $vars(pre(a)) \subseteq F^C \cup F^L$.

Due to these action behaviors, a fork factoring encapsulates a particular form of "conditional independence" between leaves. Assume a *center path* π^C, i.e., a sequence of center actions applicable to s_0. A *leaf path* is a sequence of leaf actions applicable to s_0 when ignoring preconditions on F^C. A leaf path π^L *complies* with π^C if it uses only the center preconditions supplied along π^C, i.e., if π^L can be scheduled alongside π^C so that the combined action sequence is applicable in s_0. Intuitively, fixing π^C, the compliant leaf paths are the possible leaf moves given π^C. Observe that these possible moves are independent across leaf factors F^L, i.e., for each F^L we can choose a compliant π^L independently from that choice for any other leaf factor. Hence we can search over center paths π^C only, maintaining all possible compliant paths separately for each leaf. We commit to the actual choices of compliant leaf paths only when the goal is reached.

Concretely, a *decoupled state* $s^\mathcal{F}$ is given by a center path $\pi^C(s^\mathcal{F})$. It is associated with its *center state* $ct(s^\mathcal{F})$, simply the outcome of applying $\pi^C(s^\mathcal{F})$ to $s_0[F^C]$; and with its *pricing function* $prices(s^\mathcal{F})$. The latter maps each *leaf state* s^L, i.e., each value assignment to some leaf F^L, to its *price*, defined as the cost of a cheapest leaf path that complies with $\pi^C(s^\mathcal{F})$ and ends in s^L (or ∞ if no such path exists). Pricing functions can be maintained in time low-order polynomial in the size of the individual F^L state spaces; we omit the details for space reasons. Note the word "price": $prices(s^\mathcal{F})[s^L]$ is not a cost we have already paid; rather, it is the cost we *will* have to pay in case we commit to s^L in $s^\mathcal{F}$ later on.

The *initial decoupled state* $s_0^\mathcal{F}$ results from the empty center path $\pi^C(s_0^\mathcal{F}) = \langle\rangle$. We denote by $ReachedL(s^\mathcal{F})$ the set of leaf states s^L *reachable* in $s^\mathcal{F}$, i.e., where $prices(s^\mathcal{F})[s^L] < \infty$. A *goal decoupled state* $s_\star^\mathcal{F}$ is one with a *goal center state* $ct(s_\star^\mathcal{F}) \models s_\star[F^C]$ and where, for every leaf factor $F^L \in \mathcal{F}^L$, there exists a reachable *goal leaf state* s^L, i.e., $s^L \in ReachedL(s_\star^\mathcal{F})$ such that $s^L \models s_\star[F^L]$. The actions applicable in $s^\mathcal{F}$ are those center actions whose precondition is satisfied in $ct(s^\mathcal{F})$ (recall here that we do not branch over leaf actions). Applying a to $s^\mathcal{F}$ results in $t^\mathcal{F}$ where $\pi^C(t^\mathcal{F}) := \pi^C(s^\mathcal{F}) \circ \langle a \rangle$, and $ct(t^\mathcal{F})$ as well as $prices(t^\mathcal{F})$ arise from $\pi^C(t^\mathcal{F})$ as defined above.

In our example, $ct(s_0^\mathcal{F}) = \{(t, A)\}$, and for each p_i the price of (p_i, A) is 0, that of (p_i, t) is 1, and that of (p_i, B) is ∞. Observe here that the prices represent possible package moves given the initial center state, rather than moves we have already committed to. The only action applicable to $s_0^\mathcal{F}$ in the decoupled search is the center action *drive*(t, A, B), leading to the goal decoupled state $s_\star^\mathcal{F}$ where $ct(s_\star^\mathcal{F}) = \{(t, B)\}$ and the prices are as before except that (p_i, B) now has price 2, i.e., the package goals are reachable.

Once a goal decoupled state $s_\star^\mathcal{F}$ is reached, a plan π for the input task Π can be constructed by augmenting the center path $\pi^C(s_\star^\mathcal{F})$ with compliant leaf paths ending in goal leaf states $s_\star^L$ (i.e., we now select such leaf paths, and commit to them). In our example, for each p_i we may select the compliant leaf path $\langle$*load*(t, p_i, A),*unload*$(t, p_i, B)\rangle$.

Selecting, for the plan π, the *cheapest* compliant paths ending in the goal leaf states $s_\star^L$, by construction we have $cost(\pi) = cost(\pi^C(s_\star^\mathcal{F})) + \sum_{F^L \in \mathcal{F}^L} prices(s_\star^\mathcal{F})[s_\star^L]$. If we select $s_\star^L$ with *minimal* $prices(s_\star^\mathcal{F})[s_\star^L]$, such π is optimal among the plans for Π whose center action subsequence is $\pi^C(s_\star^\mathcal{F})$. Given this, we refer to the cost of such π as the *local cost* of $s_\star^\mathcal{F}$, denoted $LocalCost(s_\star^\mathcal{F})$. We set $LocalCost(s^\mathcal{F}) := \infty$ for non-goal decoupled states $s^\mathcal{F}$.

$LocalCost(s_\star^\mathcal{F})$ is optimal for $s_\star^\mathcal{F}$ (locally optimal) but not necessarily optimal for Π (globally optimal). Indeed, it can happen that, from $s_\star^\mathcal{F}$, a *better* plan can be obtained from a descendant of $s_\star^\mathcal{F}$. This is because, with additional center actions, cheaper leaf paths may become available. For example, say in $s^\mathcal{F}$ the leaf goal *have-car* has price 1000 via the applicable leaf action *buy-car*. But if we apply a center action *get-manager-job*, then the leaf action *get-company-car* becomes applicable, reducing the leaf goal price to 0.

In contrast to the standard setting, to guarantee optimality one must therefore continue the search on goal decoupled states (GH show that standard search algorithms are easy to

adapt to this situation). The purpose of such search, trying to decrease leaf prices, differs from that of non-goal decoupled states, trying to reach the goal in the first place. Our design of strong stubborn sets for decoupled search distinguishes between the two cases.

SSS for Non-Goal Decoupled States

We show that, for non-goal decoupled states, the definition of strong stubborn sets (SSS) for planning (Alkhazraji *et al.* 2012) can be extended to decoupled search by suitable extensions of its basic components.

A SSS for a given state s is a set $T_s \subseteq \mathcal{A}$ constructed so that, for every plan π for s, at least one permutation of π starts with an action $a \in T_s$. Hence SSS are fundamentally based on the concept of "plans for a given state". That concept is trivial for classical state spaces. But in decoupled state spaces the structure of "states" $s^{\mathcal{F}}$ is more complex. GH did not require, so did not introduce, such a concept. For our purposes, the following notions will suffice.

A path $\pi^{\mathcal{F}}$ in the decoupled state space is a *decoupled plan* for $s^{\mathcal{F}}$ if it leads from $s^{\mathcal{F}}$ to a goal decoupled state. We say that $s^{\mathcal{F}}$ is *solvable* if at least one such $\pi^{\mathcal{F}}$ exists. We denote the center-action sequence underlying $\pi^{\mathcal{F}}$ by $\pi^C(\pi^{\mathcal{F}})$. The *completion plan* given $\pi^{\mathcal{F}}$, denoted $ComPlan(\pi^{\mathcal{F}})$, consists of $\pi^C(\pi^{\mathcal{F}})$ together with cheapest goal leaf paths π^L compliant with $\pi^C(s^{\mathcal{F}}) \circ \pi^C(\pi^{\mathcal{F}})$, ending in cheapest goal leaf states. In other words, $ComPlan(\pi^{\mathcal{F}})$ collects the postfix path for the center, and the complete path for each leaf. Observe that $ComPlan(\pi^{\mathcal{F}})$ is not uniquely defined, as there may be multiple suitable π^L. For our purposes, this does not matter and we assume any suitable choice of π^L. We say that $\pi^{\mathcal{F}}$ is *optimal* if $cost(ComPlan(\pi^{\mathcal{F}}))$ is minimal among all decoupled plans for $s^{\mathcal{F}}$. In our running example, assume a third location C and the road map $A \rightarrow B \rightarrow C$. Say we apply $drive(t, A, B)$ to $s_0^{\mathcal{F}}$ to obtain $s^{\mathcal{F}}$. Then $\pi^C := \langle drive(t, B, C)\rangle$ yields a decoupled plan $\pi^{\mathcal{F}}$ for $s^{\mathcal{F}}$, and $ComPlan(\pi^{\mathcal{F}})$ consists of all $load(t, p_i, A)$ actions, then π^C, then all $unload(t, p_i, C)$ actions.

Clearly, to preserve optimality, it suffices for T_s to contain at least one center action starting an optimal decoupled plan for $s^{\mathcal{F}}$. Towards identifying sets T_s qualifying for this, we will need to focus exclusively on the part of the completion plan "behind" $s^{\mathcal{F}}$. We denote this by $PostPlan(\pi^{\mathcal{F}})$, the *postfix plan*. The center action subsequence in $PostPlan(\pi^{\mathcal{F}})$ is $\pi^C(\pi^{\mathcal{F}})$. For any leaf factor F^L, say $\pi^L = \langle a_1^L, \dots, a_n^L\rangle$ is the goal leaf path for F^L in $ComPlan(\pi^{\mathcal{F}})$, traversing leaf states $\langle s_0^L, \dots, s_n^L\rangle$. Then the leaf action subsequence for F^L in $PostPlan(\pi^C)$ is defined as $\langle a_{i+1}^L, \dots, a_n^L\rangle$, where i is the highest index for which $s_i^L \in ReachedL(s^{\mathcal{F}})$. In other words, we consider the postfix of π^L not contained in $ReachedL(s^{\mathcal{F}})$.

Two notions, of completion plan *and* postfix plan, are required because postfix plans are (in contrast to the standard setting) *not* suited to define optimality. The decoupled plan leading to the cheapest postfix plan may differ from that leading to the cheapest completion plan. This is because the postfix plan ignores the price of the $s^{\mathcal{F}}$ leaf states it starts from.

The original definition of SSS in states s relies on the basic concepts of *disjunctive action landmarks*, *action interference*, *necessary enabling sets*, and *action applicability*. For a corresponding definition for decoupled states $s^{\mathcal{F}}$, the concept of action interference remains the same, but all other concepts must be extended. We start with applicability:

Definition 1 (Action Applicability)**.** *Let $s^{\mathcal{F}}$ be a decoupled state. A center action a is* applicable *in $s^{\mathcal{F}}$ if $ct(s^{\mathcal{F}}) \models pre(a)$. A leaf action a affecting leaf F^L is* applicable *in $s^{\mathcal{F}}$ if $ct(s^{\mathcal{F}}) \models pre(a)[F^C]$, and there exists a leaf state $s^L \in ReachedL(s^{\mathcal{F}})$ such that $s^L \models pre(a)[F^L]$. The set of actions applicable in $s^{\mathcal{F}}$ is denoted with $app^{dec}(s^{\mathcal{F}})$.*

Note that this definition encompasses both, center actions *and* leaf actions. This is in contrast to the decoupled search which branches only over (applicable) center actions. Thus the notion of "applicability" as per Definition 1 is different from the notion used in decoupled search. It is better suited for the definition of strong stubborn sets, lending itself to a direct extension of the original definition.

Let us next focus on the concept of necessary enabling sets. Given an action a whose preconditions are not true, a necessary enabling set should be a set of actions one of which must necessarily be applied in order to enable a. In the standard setting, such a set is trivial to obtain, by picking a precondition value not currently true and selecting all actions achieving that value. In decoupled search, this is not as easy because decoupled states do not assign unique values to leaf-factor state variables. We adjust the concept as follows:

Definition 2 (Decoupled Necessary Enabling Set)**.** *Let $s^{\mathcal{F}}$ be a decoupled state, and let a be an inapplicable action $a \notin app^{dec}(s^{\mathcal{F}})$. An action set A is a* decoupled necessary enabling set *for a in $s^{\mathcal{F}}$ if either of the following cases holds:*

(i) $A = \{a' \in \mathcal{A} \mid eff(a')[v] = pre(a)[v]\}$ *where* $v \in vars(pre(a)) \cap F^C$ *s.t.* $ct(s^{\mathcal{F}})[v] \neq pre(a)[v]$.

(ii) $A = \{a' \in \mathcal{A} \mid eff(a')[v] = pre(a)[v]\}$ *where* $v \in vars(pre(a)) \setminus F^C$ *s.t., for all* $s^L \in ReachedL(s^{\mathcal{F}})$, *we have* $s^L \not\models pre(a)[v]$.

(iii) $A = \bigcup_{v \in V}\{a' \in \mathcal{A} \mid eff(a')[v] = pre(a)[v]\}$ *where* $V \neq \emptyset$ *is the set of all* $v \in vars(pre(a))$ *s.t. exists* $s^L \in ReachedL(s^{\mathcal{F}})$ *with* $s^L \not\models pre(a)[v]$.

Case (i) in this definition corresponds to the standard setting, where A are the achievers of an open precondition on the center, whose assignment is fixed in $s^{\mathcal{F}}$. Case (ii) captures the situation where a leaf precondition is false in *all* reachable leaf states. Case (iii) is relevant because a leaf action may have several preconditions, each satisfied by some reachable leaf state, but not all satisfied jointly in any reachable leaf state. We then collect the achievers of preconditions open in *any* reachable leaf state. Clearly, in every case at least one $a' \in A$ must be used by any postfix plan for $s^{\mathcal{F}}$ that contains a. At least one of the cases must apply as $a \notin app^{dec}(s^{\mathcal{F}})$. For center actions, only case (i) is possible. For leaf actions, we first test (ii), then (iii), and finally (i), the motivation being to focus on the open center preconditions "closest" to $s^{\mathcal{F}}$.

We finally need to adjust the concept of disjunctive action landmarks. Given our notion of postfix plans, this is direct:

Definition 3 (Decoupled Disjunctive Action Landmark)**.** *Let $s^{\mathcal{F}}$ be a non-goal decoupled state. An action set L is a* decoupled disjunctive action landmark *for $s^{\mathcal{F}}$ if, for all decoupled plans π^C for $s^{\mathcal{F}}$, we have $PostPlan(\pi^C) \cap L \neq \emptyset$.*

In our implementation, we find decoupled disjunctive action landmarks simply in terms of a necessary enabling set for the goal condition $s_\star$, i.e., exactly as in Definition 2 but using $s_\star$ as the precondition of a hypothetical action a.

The last basic concept we need is the standard notion of *interference* between pairs of actions. We say that a and a' interfere if ex. v s.t. $\mathit{eff}(a')[v] \neq \mathit{eff}(a)[v]$ or $\mathit{eff}(a)[v] \neq \mathit{pre}(a')[v]$ or $\mathit{eff}(a')[v] \neq \mathit{pre}(a)[v]$. Decoupled strong stubborn sets are now defined as follows:

Definition 4 (DSSS for Non-Goal Decoupled States)**.** *Let $s^{\mathcal{F}}$ be a non-goal decoupled state. An action set T_s is a* decoupled strong stubborn set (DSSS) *for $s^{\mathcal{F}}$ if the following conditions hold:*

(i) T_s contains a decoupled disjunctive action landmark.

(ii) For all actions $a \in T_s$ and $a \notin app^{dec}(s^{\mathcal{F}})$, T_s contains a decoupled necessary enabling set for a.

(iii) For all center actions $a \in T_s$ and $a \in app^{dec}(s^{\mathcal{F}})$, T_s contains all actions that interfere with a.

Thanks to the adapted basic concepts, this definition mirrors the original one (Alkhazraji *et al.* 2012). Intuitively, condition (i) ensures that T_s makes progress to the goal; condition (ii) ensures that T_s backchains all the way to the current state; condition (iii) ensures that, if we branch over a, then we also branch over all actions that may be in conflict with a. All three conditions are identical to the respective original one, modulo the adapted basic concepts. The single exception is the restriction to center actions in condition (iii). We do not need to include interfering actions for applicable leaf actions. That is so because postfix plans do not contain applicable leaf actions anyhow: everything that can be done using such actions is already reachable in $s^{\mathcal{F}}$.

By adapting the proof arguments from the standard setting, one can show that DSSS preserve optimality:

Theorem 1. *Let $s^{\mathcal{F}}$ be a solvable non-goal decoupled state. Let T_s be a DSSS in $s^{\mathcal{F}}$. Then T_s contains a center action that starts an optimal decoupled plan for $s^{\mathcal{F}}$.*

The proof considers any decoupled plan $\pi^{\mathcal{F}}$ for $s^{\mathcal{F}}$. Denote $\pi = \langle a_1, \dots, a_m \rangle = PostPlan(\pi^{\mathcal{F}})$, and let i be the smallest index so that $a_i \in T_s$. For the same reasons as shown in the original proof for SSS (Alkhazraji *et al.* 2012), such a_i exists, must be applicable, and – together with the fact that a_i must be a center action, as $PostPlan(s^{\mathcal{F}})$ does not contain any applicable leaf actions – can be moved to the front of π.

SSS for Goal Decoupled States

Say we are facing a goal decoupled state $s^{\mathcal{F}}_\star$. Instead of actions required for reaching the goal, we need to capture actions required to reduce the leaf-goal prices. One may consider to define landmarks relative to the decoupled plans reaching states $t^{\mathcal{F}}_\star$ where $LocalCost(t^{\mathcal{F}}_\star) < LocalCost(s^{\mathcal{F}}_\star)$, and then re-use the remainder of Definition 4 unchanged. Indeed, this was our first solution attempt. The problem is that the landmark actions may pertain to leaf states already reached, only at non-optimal prices; and then we may miss the actions required to reduce those prices.

To illustrate, say that, as before, we have a leaf action *buy-car* (cost 1000) applicable to $s^{\mathcal{F}}$, and a center action *get-manager-job* which enables leaf action *get-company-car* (cost 0). However, now the leaf goal is not *have-car*, but *be-at-NYC* for which another leaf action *drive-car* is needed. Then $\{$*drive-car*$\}$ is a landmark: Any optimal completion plan for $s^{\mathcal{F}}$ has to use this action behind $s^{\mathcal{F}}$, i.e., after applying another center action. But *drive-car* is applicable in $s^{\mathcal{F}}$, so Definition 4 would stop here, and T_s would not contain *get-company-car*. In other words, the notion of necessary enabling sets is suited to reachability but is not suited to capture what's needed to decrease prices.

We tackle this situation through a notion of *frontier* actions, required to make *any* progress on the prices:

Definition 5 (Frontier Action)**.** *Let $s^{\mathcal{F}}_\star$ be a decoupled goal state, and let a be a leaf action affecting leaf F^L. We say that a is a* frontier action *in $s^{\mathcal{F}}_\star$ if (i) $a \notin app^{dec}(s^{\mathcal{F}}_\star)$; and (ii) there exists a leaf state $s^L \in ReachedL(s^{\mathcal{F}}_\star)$ such that $s^L \models pre(a)[F^L]$, and, denoting the outcome of applying a to s^L with t^L, $prices(s^{\mathcal{F}}_\star)[s^L] + cost(a) < prices(s^{\mathcal{F}}_\star)[t^L]$. The* frontier *of $s^{\mathcal{F}}_\star$ is the set of all frontier actions in $s^{\mathcal{F}}_\star$.*

In words, the frontier consists of those leaf actions that are not currently applicable, but enabling whose center precondition would result in a reduced price for at least one leaf state. This set of actions now takes the role of the landmark:

Definition 6 (DSSS for Goal Decoupled States)**.** *Let $s^{\mathcal{F}}_\star$ be a goal decoupled state. An action set T_s is a* decoupled strong stubborn set (DSSS) *for $s^{\mathcal{F}}_\star$ if the following conditions hold:*

(i) T_s contains the frontier of $s^{\mathcal{F}}_\star$.

(ii) For all actions $a \in T_s$ and $a \notin app^{dec}(s^{\mathcal{F}}_\star)$, T_s contains a decoupled necessary enabling set for a.

(iii) For all center actions $a \in T_s$ and $a \in app^{dec}(s^{\mathcal{F}}_\star)$, T_s contains all actions that interfere with a.

Consider now a state $s^{\mathcal{F}}_\star$ where $\langle\rangle$ is not an optimal decoupled plan, i.e., we can find a better plan below $s^{\mathcal{F}}_\star$. Consider any decoupled plan $\pi^{\mathcal{F}}$ leading to $t^{\mathcal{F}}_\star$ where $LocalCost(t^{\mathcal{F}}_\star) < LocalCost(s^{\mathcal{F}}_\star)$. Then $ComPlan(\pi^{\mathcal{F}})$ contains at least one frontier action a_F, intuitively because these actions are needed to decrease prices relative to $s^{\mathcal{F}}_\star$. By construction, a_F has a center precondition not satisfied in $s^{\mathcal{F}}_\star$. Therefore, with the inclusion of necessary enabling sets, we get that T_s must contain an applicable center action a of $\pi^{\mathcal{F}}$. For the same reasons as before we can move a to the front, proving that DSSS as per Definition 6 preserve optimality:

Theorem 2. *Let $s^{\mathcal{F}}_\star$ be a goal decoupled state for which $\langle\rangle$ is not an optimal decoupled plan. Let T_s be a decoupled strong stubborn set for $s^{\mathcal{F}}_\star$. Then T_s contains a center action that starts an optimal decoupled plan for $s^{\mathcal{F}}_\star$.*

Observe that $Frontier(s^{\mathcal{F}})$ may be empty. In that case, the DSSS will be empty, too. This is valid because, in this case, necessarily $\langle\rangle$ is an optimal decoupled plan for $s^{\mathcal{F}}_\star$, i.e.,

no better plan can be found below $s_\star^{\mathcal{F}}$ and the search can stop.

Exponential Separations

Before proceeding to the empirical part of our research, let us state some basic theoretical facts evaluating the power of DSSS. We say that a search space reduction method X is *exponentially separated* from a method Y if there exists a parameterized example family $\mathcal{F}$ such that, on $\mathcal{F}$, X yields an exponentially stronger reduction than Y.

Decoupled search and SSS are complementary in that each is exponentially separated from the other:

Theorem 3. *Fork-decoupled search is exponentially separated from SSS, and vice versa.*

Our running example with locations A and B is a suitable family $\mathcal{F}$ for the first claim. There are only 3 reachable decoupled states ($s_0^{\mathcal{F}}$; drive to B; drive back). But SSS do not yield any pruning because, in any state s, to make progress to the goal, T_s must include an applicable *(un)load* action; which interferes with the applicable *drive* action; which in turn interferes with all applicable *(un)load* actions. The opposite claim follows from examples, e.g. IPC Parcprinter, with no fork factoring but strong SSS pruning.

Trivially, DSSS is exponentially separated from each of fork-decoupled search and SSS, simply because DSSS naturally generalizes each of these components, so we can use the same families $\mathcal{F}$ as in Theorem 3. As a much stronger testimony to the power of DSSS, there are cases where it is exponentially separated from *both* its components:

Theorem 4. *There exists a parameterized example family $\mathcal{F}$ such that, on $\mathcal{F}$, DSSS yields an exponentially stronger reduction than both, fork-decoupled search and SSS.*

Two suitable families $\mathcal{F}$ arise from simple modifications of our running example. First, say we have M trucks and $N * M$ packages, where each truck t_i is associated with a group of N packages that only t_i can transport. The number of reachable decoupled states is exponential in M because all trucks must be in the center factor. The SSS-pruned reachable standard state space has size exponential in N because including an *(un)load* action into T_s necessitates, due to interference via the truck move as above, to include *all* applicable *(un)load* actions for the respective package group. However, *in decoupled search with DSSS pruning, there are only M reachable states*. This is because the two sources of pruning power combine gracefully. Decoupling gets rid of the blow-up in N (the packages within a group become independent leaves), while DSSS gets rid of the blow-up in M (only a single truck is committed to at a time).

In our second example, DSSS even is exponentially *more* than the sum of its components: stubborn sets have exponentially more impact on the decoupled search space than on the standard one. Say we have N packages and M trucks (where every truck may transport every package). Then decoupled search blows up in M, and SSS does not do anything because any package may require any truck. Applying DSSS to decoupled search, no truck move is pruned in $s_0^{\mathcal{F}}$. However, after applying any one $drive(t_i, A, B)$ action, all package prices are the cheapest possible ones, the frontier is empty, and DSSS stops the search. So, again, there are only M reachable states. As we shall see next, similar phenomena seem to occur in the standard IPC Logistics benchmarks.

Experiments

We extended GH's implementation of fork-decoupled search in FD (Helmert 2006). To extract the fork factorings, we use GH's method. It computes the strongly connected components (SCCs) of the causal graph, and, arranging the acyclic graph of SCCs with roots "at the top" and leaves "at the bottom", greedily finds a "horizontal line" through that graph. The part above the line becomes the center, each weakly connected component below the line becomes a leaf. The technique abstains if there is ≤ 1 leaf, the rationale being that decoupling pays off mainly through avoiding enumeration across > 1 leaves. We show results for those benchmarks on which the technique does not abstain. From the International Planning Competition (IPC) STRIPS benchmarks ('98–'14), this is the case for instances from 12 domains.

We focus here on optimal planning, the main purpose of the optimality-preserving pruning via strong stubborn sets. We run A* with a blind heuristic as a measure of search space size, and with LM-cut (Helmert and Domshlak 2009) as a representative of the state of the art, using GH's method (Fork-Decoupled A*) to adopt these techniques for decoupled search. We compare decoupled search with DSSS pruning (simply referred to as "DSSS" in what follows) against decoupled search without that pruning ("DS" in what follows). We furthermore compare against A* in the standard state space without pruning ("A*" in what follows), and with SSS pruning ("SSS" in what follows). All experiments are run on a cluster of Intel E5-2660 machines running at 2.20 GHz, with time (memory) cut-offs of 30 minutes (4 GB).

		Blind Heuristic				LM-cut			
Domain	#	A*	SSS	DS	DSSS	A*	SSS	DS	DSSS
Driverlog	20	7	7	11	11	13	13	13	13
Logistics'00	28	10	10	22	24	20	20	28	28
Logistics'98	35	2	2	4	5	6	6	6	6
Miconic	145	50	45	35	36	136	136	135	135
NoMystery	20	8	7	17	15	14	14	20	19
Pathways	29	3	3	3	3	4	4	4	4
Rovers	40	6	7	7	9	7	9	9	11
Satellite	36	6	6	6	6	7	11	7	11
TPP	27	5	5	23	22	5	5	18	18
Wood'08	13	4	6	5	7	6	11	10	11
Wood'11	5	0	1	1	2	2	5	4	5
Zenotravel	20	8	7	11	11	13	13	13	13
$\sum$	418	109	106	145	151	233	247	267	274

Table 1: Coverage (number of instances solved).

Table 1 shows coverage results. The most important comparison for our purposes here is that between DSSS vs. DS, i.e., the direct benefit our pruning technique yields over the baseline search. DSSS is rarely worse (NoMystery -2 and TPP -1 for blind search, only NoMystery -1 for LM-cut). It is often better (6 domains for blind, 4 domains for LM-cut), and consequently is better, though not dramatically better, in the overall. Comparing to A* and SSS, we see that

Blind Heuristic									LM-cut								
	Expansions				Runtime					Expansions				Runtime			
Inst	A*	SSS	DS	DSSS	A*	SSS	DS	DSSS	Inst	A*	SSS	DS	DSSS	A*	SSS	DS	DSSS
Logistics'00																	
p6-9	368109	368109	30	9	2.2	5.2	0.0	0.0	p12-0	116544	116544	149	98	132.3	141.8	0.2	0.1
p12-0	–	–	8101	882	–	–	15.1	0.2	p14-0	–	–	4130	2193	–	–	17.1	6.6
p12-1	–	–	22644	1338	–	–	46.4	0.3	p14-1	–	–	8263	4726	–	–	42.3	17.9
p14-0	–	–	–	197855	–	–	–	605.8	p15-0	–	–	41259	15977	–	–	280.3	62.5
p14-1	–	–	–	324152	–	–	–	1256.9	p15-1	–	–	11710	5978	–	–	59.6	21.7
Logistics'98																	
p1	–	–	75954	15404	–	–	325.48	14.2	p1	12634	12634	555	379	13.6	15.2	1.0	0.5
p5	–	–	–	20410	–	–	–	45.24	p31	56	56	12	12	0.0	0.0	0.0	0.0
p31	133855	133855	586	224	1.31	3.53	0.16	0.04	p32	108	47	20	15	0.0	0.0	0.0	0.0
p32	218003	218003	368	124	1.39	3.3	0.04	0.01	p33	92692	92692	388	104	85.8	94.2	0.4	0.1
p33	–	–	3550	592	–	–	1.43	0.2	p35	1636	1636	360	360	11.8	12.5	4.8	1.8
Pathways																	
p2	2916	531	2366	489	0.0	0.0	0.0	0.0	p3	98	64	98	66	0.0	0.0	0.0	0.0
p3	53603	2252	16030	609	0.3	0.0	0.7	0.0	p4	189	150	189	150	0.0	0.0	0.1	0.0
p4	300600	10331	31903	2131	3.8	0.2	2.6	0.1	p5	46402	6675	27346	3989	51.8	6.2	39.9	4.1
Rovers																	
p5	7.52M	213647	152871	6861	71.5	5.3	15.2	0.5	p5	71222	4562	9533	1154	9.7	0.5	2.2	0.2
p6	–	1.20M	6.28M	28693	–	21.0	763.8	1.9	p8	–	–	1.16M	1.00M	–	–	896.1	630.5
p7	32.78M	25.19M	522185	406676	301.1	489.6	43.8	35.2	p9	–	892779	–	8573	–	205.5	–	3.8
p9	–	–	–	397564	–	–	–	55.8	p12	19195	11788	8915	4915	9.9	5.1	6.1	2.6
p12	–	–	–	1.52M	–	–	–	278.8	p14	–	–	–	780716	–	–	–	481.7
Satellite																	
p2	1539	1471	303	249	0.0	0.0	0.0	0.0	p7	95606	3204	77253	10735	120.8	4.9	156.7	12.3
p3	13243	5839	1484	857	0.1	0.1	0.1	0.1	p9	–	3722	–	40514	–	35.1	–	194.1
p4	274070	14510	27706	16225	3.1	0.3	19.0	7.9	p10	–	172718	–	–	–	1399.1	–	–
p5	22.98M	3.01M	364513	217733	636.5	106.2	181.1	149.6	p11	–	0	–	0	–	9.2	–	16.2
p6	19.81M	142382	2.17M	121935	402.7	6.3	1358.1	40.2	p18	–	8366	–	4665	–	98.2	–	140.5
Woodworking'08																	
p1	9797	1002	1	1	0.1	0.0	0.0	0.0	p7	32418	177	224	46	601.0	1.7	5.4	1.2
p2	23287	70	0	0	0.2	0.0	0.0	0.0	p8	–	694	–	5268	–	10.8	–	61.8
p9	–	1.65M	–	30851	–	88.9	–	4.5	p9	–	5157	1103	43	–	5.7	9.3	0.3
p16	–	–	–	1.78M	–	–	–	223.1	p24	9868	425	615	168	19.8	0.4	1.3	0.3
p24	–	137867	1210202	21721	–	5.2	120.6	1.7	p30	–	308	525	62	–	50.7	463.6	18.1
Woodworking'11																	
p5	–	137867	1.21M	21721	–	5.4	132.4	1.7	p5	9868	425	615	168	19.9	0.4	1.2	0.3
p12	–	–	–	1.78M	–	–	–	220.7	p12	–	18317	22072	1920	–	30.3	118.9	5.5
									p13	–	0	0	0	–	0.1	0.2	0.1
									p16	32418	177	224	46	621.2	1.7	5.6	1.1
									p19	–	694	–	5268	–	10.4	–	61.5

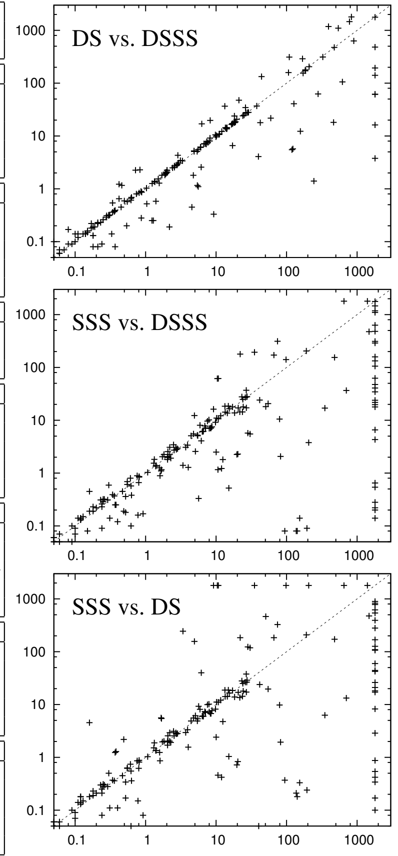


Figure 1: Runtime, and expansions to prove optimality (before last f-layer in A*). Table: Per-instance data on selected IPC instances "Inst" (see text). "M": million, "–": out of time or memory. Scatter plots: Runtime with LM-cut, for all pairs "X vs. Y" of non-baseline configurations. X on x-axis, Y on y-axis, time-out 1800 seconds inserted for unsolved instances.

DSSS improves DS whenever (i.e., in all domains where) SSS improves A*. Whenever SSS outperforms DS, DSSS fully makes up for this advantage: the per-domain coverage of DSSS dominates that of SSS. The single exception to the latter is Miconic, where DSSS just inherits the weakness (runtime overhead at not much search gain) of decoupled search.

Figure 1 shows fine-grained performance data. Consider first the scatter plots. The plot at the top reveals that DSSS often improves over DS, up to 2 orders of magnitude on commonly solved instances, while bad cases are consistently limited to a moderate overhead. The plot SSS vs. DS shows that, without pruning, decoupled search is in the advantage yet also incurs several bad cases. We see in the plot SSS vs. DSSS that, with DSSS pruning, this risk mostly disappears.

The table in Figure 1 shows data for those domains where DSSS sometimes reduces expansions relative to DS (we discuss the other domains below). For each of blind search and LM-cut, from the instances solved by at least one method, we selected at most 5, namely the most challenging ones (largest expansions under standard A*). Where these did not include an instance solved by all methods, to exemplify the cross-comparison we included the most challenging such instance.

As the table shows, on those domains where DSSS does yield pruning, it consistently improves over DS, both in expansions and runtime, for both blind search and LM-cut. The behavior in Logistics is especially remarkable. On the standard state space, SSS yields little or no reduction, while in the decoupled state space, DSSS yields strong reductions. This establishes a practical case of DSSS being more than the sum of its components. Compared to SSS, decoupled search with DSSS is superior in Logistics, Pathways, and Rovers, and is inferior in Satellite; the picture in Woodworking is mixed.

On the domains where DSSS does *not* reduce expansions (Driverlog, Miconic, NoMystery, TPP, and Zenotravel), a runtime overhead is incurred. For blind search, we get slow-down factors up to 218.5 in NoMystery, 26.9 in TPP, and 4.8 in the other 3 domains. This is due to the small per-state search effort in blind search, relative to which computing a DSSS can consume substantial runtime. For the state-of-the-art search using LM-cut, where per-state effort is much higher, the overhead is small. The maximum (geometric mean) slow-down factor is 1.3 (1.1) for Driverlog, 2.0 (1.0) for Miconic, 3.1 (2.2) for NoMystery, 2.0 (1.3) for TPP, and 2.0 (1.1) for Zenotravel. Using a simple "safety belt" which switches DSSS off after 1000 expansions if no action was pruned, the slow-down disappears in almost all cases.

Conclusion

We have shown that fork-decoupled search and strong stubborn sets combine gracefully in theory, and that the combination can yield good results in practice. Our next step will be to extend decoupled strong stubborn sets to star-topology decoupling as per Gnad and Hoffmann (2015b). More generally, decoupled search is a new paradigm that, presumably, can be fruitfully combined not only with (heuristic search and) strong stubborn sets, but also with other search techniques like symmetry reduction or symbolic representations.

Acknowledgments

Daniel Gnad was partially supported by the German Research Foundation (DFG), as part of project grant HO 2169/6-1, "Star-Topology Decoupled State Space Search". Martin Wehrle was supported by the Swiss National Science Foundation (SNSF) as part of the project "Automated Reformulation and Pruning in Factored State Spaces (ARAP)".

References

Yusra Alkhazraji, Martin Wehrle, Robert Mattmüller, and Malte Helmert. A stubborn set algorithm for optimal planning. In Luc De Raedt, Christian Bessiere, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter Lucas, editors, *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI 2012)*, pages 891–892. IOS Press, 2012.

Eyal Amir and Barbara Engelhardt. Factored planning. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, pages 929–935. Morgan Kaufmann, 2003.

Christer Bäckström and Bernhard Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

Ronen I. Brafman and Carmel Domshlak. Structure and complexity in planning with unary operators. *Journal of Artificial Intelligence Research*, 18:315–349, 2003.

Ronen Brafman and Carmel Domshlak. On the complexity of planning for agent teams and its implications for single agent planning. *Artificial Intelligence*, 198:52–71, 2013.

Eric Fabre, Loïg Jezequel, Patrik Haslum, and Sylvie Thiébaux. Cost-optimal factored planning: Promises and pitfalls. In Ronen Brafman, Héctor Geffner, Jörg Hoffmann, and Henry Kautz, editors, *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, pages 65–72. AAAI Press, 2010.

Daniel Gnad and Jörg Hoffmann. Beating LM-cut with h^{max} (sometimes): Fork-decoupled state space search. In Ronen Brafman, Carmel Domshlak, Patrik Haslum, and Shlomo Zilberstein, editors, *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, pages 88–96. AAAI Press, 2015.

Daniel Gnad and Jörg Hoffmann. From fork decoupling to star-topology decoupling. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search (SoCS 2015)*, pages 53–61. AAAI Press, 2015.

Daniel Gnad, Martin Wehrle, and Jörg Hoffmann. Decoupled strong stubborn sets (technical report). Technical report, Saarland University, 2016. Available at `http://fai.cs.uni-saarland.de/hoffmann/papers/ijcai16a-tr.pdf`.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 162–169. AAAI Press, 2009.

Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Peter Jonsson and Christer Bäckström. Incremental planning. In Malik Ghallab and Alfredo Milani, editors, *New Directions in AI Planning: EWSP '95 — 3rd European Workshop on Planning*, volume 31 of *Frontiers in Artificial Intelligence and Applications*, pages 79–90, Amsterdam, 1995. IOS Press.

Elena Kelareva, Olivier Buffet, Jinbo Huang, and Sylvie Thiébaux. Factored planning using decomposition trees. In Manuela M. Veloso, editor, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, pages 1942–1947, 2007.

Craig A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets (APN 1989)*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer-Verlag, 1989.

Martin Wehrle and Malte Helmert. About partial order reduction in planning and computer aided verification. In Lee McCluskey, Brian Williams, José Reinaldo Silva, and Blai Bonet, editors, *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*. AAAI Press, 2012.

Martin Wehrle and Malte Helmert. Efficient stubborn sets: Generalized algorithms and selection strategies. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, pages 323–331. AAAI Press, 2014.

Optimal Solitaire Game Solutions using A* Search and Deadlock Analysis

Gerald Paul
Boston University
Boston, Massachusetts, USA
gerryp@bu.edu

Malte Helmert
University of Basel
Basel, Switzerland
malte.helmert@unibas.ch

Abstract

We propose and implement an efficient method for determining optimal solutions to such skill-based solitaire card games as Freecell and King Albert solitaire. We use A* search together with an admissible heuristic function that is based on analyzing a directed graph whose cycles represent deadlock situations in the game state. To the best of our knowledge, ours is the first algorithm that efficiently determines optimal solutions for Freecell games. We believe that the underlying ideas should be applicable not only to games but also to other classical planning problems which manifest deadlocks.

1 Introduction

Games have always been a fertile ground for advancements in computer science, operations research and artificial intelligence. Solitaire card games, and Freecell in particular, have been the subject of study in both the academic literature (Elyasaf, Hauptman, and Sipper 2011; 2012; Sipper and Elyasaf 2014; Long and Fox 2000; Bacchus 2001; Fox and Long 2001; Helmert 2003; Hoffmann 2005; Hoffmann and Nebel 2001; Hoffmann, Porteous, and Sebastia 2004; Morris, Tarassenko, and Kenward 2005; Pecora and Cesta 2003; Russell and Norvig 2003), where they are used as a benchmark for planning heuristics, and in popular literature (Fish 2015; Heineman 2015; FreeCell solutions 2015; Keller 2015; PySolFC 2015; Levin 2008; Van Noorden 2006; Mlot 2015).

Our work applies to *skill-based* solitaire games in which all cards are dealt face up. For these games, after the initial deal, there is no element of chance involved. Examples of such games include Freecell, King Albert, Bakers dozen, and Eight-off (Morehead and Mott-Smith 1983). Skill-based solitaire games are examples of *classical planning* problems (Ghallab, Nau, and Traverso 2004).

The Freecell solitaire game was introduced by Microsoft as a free desktop game in early versions of the Windows operating system. The rules of Freecell are described in Appendix A. While our examples and solution results are for Freecell, they apply to a large class of skill-based solitaire games.

We use Freecell because it is the most widely played and analyzed skill solitaire card game with free on-line, desktop and mobile versions of the game. Freecell games are denoted by the randomization seed that produces them in the Windows implementation of the game. Because the randomization algorithm for Freecell deals is public (Horne 2015), given a seed the random deals are reproducible, so comparisons can be made with other work. While Freecell differs in detail from other skill solitaire games, such concepts as foundation cells to which cards must ultimately be moved, and a tableau of columns of cards is common to many skill solitaire games. Freecell has been shown to be NP-hard (Helmert 2003) and thus provides a demanding test of our approach.

There are a number of Freecell computer solvers available which provide solutions to any Freecell deal (Elyasaf, Hauptman, and Sipper 2011; Fish 2015; Heineman 2015; Keller 2015; PySolFC 2015). However, we know of no work which provides provably *optimal* solutions to solitaire games. We consider a solution optimal if no other solution exists which requires a smaller number of moves.

One of the defining attributes of such skill-based games as Freecell is that *deadlocks*[1] are present and, in order to resolve the deadlocks, actions are required that do not contribute directly to reaching the goal state. Deadlock has long been recognized as a feature that makes finding optimal solutions to planning problems hard (Gupta and Nau 1992). When the state of the game is appropriately mapped to a directed graph, the deadlocks are represented by cycles of the graph.

A key insight of this work is that very strong admissible heuristic functions for Freecell can be constructed by analyzing these deadlock cycles. Graph analysis has been employed in analysis of problem complexity (Gupta and Nau 1992) and planning heuristics (Helmert 2004). One of the main contributions of this work is that we show how it can be used to optimally solve a highly popular class of puzzles that have so far defied optimal solution.

In the following sections, we review the A* algorithm, describe our approach, and presents results of our solver implementation. We conclude by discussing connections to recent research in classical planning, future research directions and open questions towards the end of this paper. While our algorithmic contributions and experimental evaluation are fo-

[1] In general, a deadlock situation exists when an action, A, cannot be taken until another action, B, is taken but action B, cannot be taken until action A is taken (and the generalization to circular waiting of multiple actions.

cused on Freecell, we believe that the key insights underlying the deadlock heuristic have much wider applicability within and outside of classical planning.

2 A* Search Algorithm

The A* search algorithm (Hart, Nilsson, and Raphael 1968) uses a best-first search and finds a least-cost path from a given initial state to the goal state. As A* traverses the state space, it builds up a tree of partial paths. The leaves of this tree (called the *open set*) are stored in a priority queue that orders the leaf states by a cost function:

$$f(n) = g(n) + h(n). \tag{1}$$

Here, $g(n)$ is the known cost of getting from the initial state to state n. $h(n)$ is a heuristic estimate of the cost to get from n to the goal state. For the algorithm to find the actual least cost path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the goal state. Roughly speaking, the closer the heuristic estimate is to the actual cost of reaching the goal state, the more efficient the algorithm.[2]

In our case, $g(n)$ is simply the number of moves that have been made to reach the state n and $h(n)$ is an estimate of the number of moves to reach the solution of the game from state n.

3 Freecell Heuristics for A*

The simplest non-trivial heuristic is $52 - m_f(n)$ where $m_f(n)$ is the number of cards in the foundation in state n. This estimate, however, is extremely optimistic because some cards are usually blocked from movement to the foundation. The simplest example of this is a column where a card of a given suit and rank is higher[3] in the column than a card of the same suit and greater rank. Until the greater rank card is moved to a temporary location in a free cell or another tableau cell, the lower rank card cannot be moved to the foundation. Then later, the greater rank card may be able to be moved to the foundation. Thus, a more robust heuristic is

$$h(n) = 52 - m_f(n) + m_e(n) \tag{2}$$

where $m_e(n)$ is an optimistic estimate of the number of moves to temporary locations that must be made to remove these blocking or deadlock situations.

There are more complicated deadlock situations than the example above. With the mapping of the game state to a directed graph described in the next section, we can associate all deadlock situations with cycles in the graph. The deadlock situations are removed when all cycles are eliminated; a cycle is eliminated when one or more edges of the cycle are removed. Now, the only way to remove an edge is to move a card and moving a card cannot remove more than one edge. Thus, the number of remaining moves must be at least as great as the number of moves to eliminate these cycles. For this reason, we take $m_e(n)$ to be an optimistic estimate of the number of edges which must be removed to eliminate all cycles. Note that this estimate may still not be an exact estimate of the number of remaining moves needed to win because the use of temporary locations is limited by the availability of open free cells, empty cells in the tableau, or a column to which the card can be moved to extend a cascade. Also note that we can use for $m_e(n)$ an estimate of the number of edges needed to remove a consistent *subset* of all cycles. This allows for performance tuning the implementation as discussed in Section 10.

4 Solitaire State to Directed Graph Mapping

We map the solitaire game layout to a directed graph as follows:

- We treat each card as a node of the graph.
- We create a directed edge from each card to the card of next lower rank of the same suit (e.g., from the 8♡ to the 7♡). We call these edges *dependency edges* because being able to move a card to the foundation depends on the card of next lower rank of the same suit being in the foundation. Dependency edges are permanent; they are never removed and are not affected when a card is moved. We define the *suit of a dependency edge* as the suit of the cards to which the edge is incident.
- We create a directed edge from each card in the tableau to the card below it in the tableau (if any). We denote these edges *blocking edges* because a card in the tableau is blocked from being moved to the foundation unless it is the exposed card (lowest card) in the tableau column. Blocking edges are removed and added when a move is made to reflect the new state of the game.

An example of dependency and blocking edges that are part of a cycle is shown in Fig. 1.

5 Cycle Determination

At first glance, the task of dealing with the cycles of the created by the mapping is daunting. For example, there are 26133 unique cycles in the graph created from mapping Freecell game #1 .

However, we can reduce the number of cycles to be considered by eliminating *redundant cycles*. A cycle c_1 is redundant if the set of blocking edges of any other cycle c_2 is a subset of the set of edges of c_1, in which case the removal of any edge in cycle c_2 results not only in the destruction of c_2 but also c_1.

Here we describe how to construct all non-redundant cycles. The approach depends on the fact that all dependency edges are always present. So from any card in a suit to any card of lower rank of the same suit we can always create a path that does not contain any blocking edges.

For conciseness, let us define a cycle that includes dependency edges of q different suits as a *q-suit cycle*. Now, first

[2] While not universally true (Holte 2010), the rule "more accurate heuristic = lower search effort" is generally a good approximation of reality.

[3] Throughout the paper, we use "higher" and "lower" to refer to the usual *visual* representation of card columns in solitaire games. For example, the "lowest" card in the leftmost column of Fig. 1 is 6♠. This is the only card in the column that may be moved directly. To move any other cards, the cards below them must first be moved out of the way.

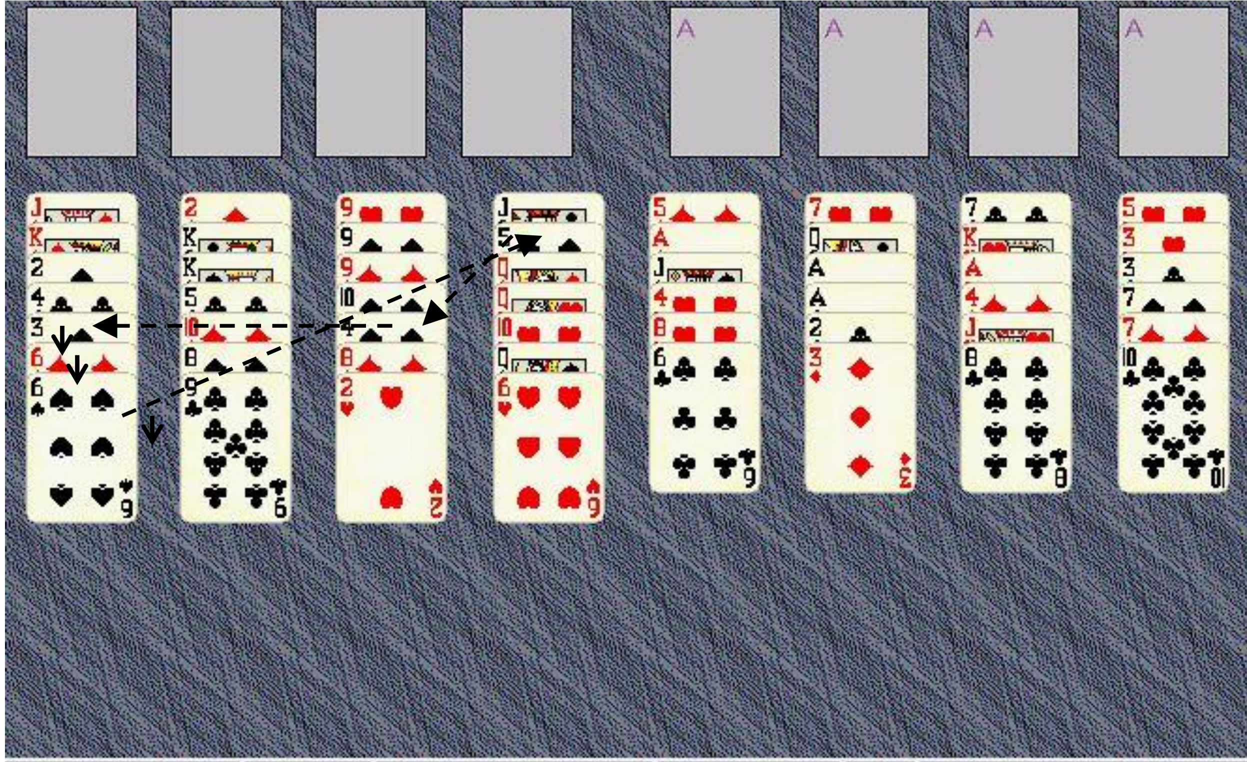


Figure 1: Initial state of Microsoft Freecell game #1. A cycle consisting of blocking edges (solid arrows) $3\spadesuit \rightarrow 6\diamondsuit, 6\diamondsuit \rightarrow 6\spadesuit$ and dependency edges (dashed lines) $6\spadesuit \rightarrow 5\spadesuit, 5\spadesuit \rightarrow 4\spadesuit, 4\spadesuit \rightarrow 3\spadesuit$ is shown.

consider cycles in which all dependency edges are of the same suit (1-suit cycles), an example of which is shown in Fig. 2. The blocking edges in the cycle in Fig. 2(a) are a subset of the blocking edges in the cycle in Fig. 2(b). So the cycle in Fig. 2(b) is a redundant cycle. We can infer the rule that if all dependency edges of a cycle are of the same suit, we need only consider cycles in which the dependency edges of the same suit in the cycle are consecutive (i.e., not interrupted by blocking edges). This rule implies that all blocking edges in a single-suit cycle are in the same tableau column.

Now consider cycles in which the dependency edges include cards of two suits (2-suit cycles), an example of which is shown in Fig. 3. Again, the cycle in Fig. 3(b) is redundant because blocking edges in the cycle in Fig. 3(a) are a subset of the blocking edges of the cycle in Fig. 3(b). Generalizing to cycles containing dependency edges of any number of suits, we can infer that

- if the dependency edges of any suit in the cycle are not consecutive, (i.e., are interrupted by blocking edges) the cycle is redundant, and that
- for a cycle containing dependency edges of q suits, the blocking edges of non-redundant cycles are contained in at most q tableau columns.

Based on the above, we can construct non-redundant q-suit cycles, $(q = 1, 2, 3, 4)$, by considering $O(t^q)$ combinations of the tableau columns, where t is the number of tableau columns (8 for Freecell). Pseudo code for identification of 1-suit and 2-suit cycles is presented in Appendix B.

6 Relevant Cycle Edges

In considering edges for removal from a cycle, not all edges must be considered. Clearly, only blocking edges (as opposed to dependency edges) must be considered, since dependency edges are never removed.

Also, in solitaire, cards higher in a column than a given card cannot be moved before the given card is moved. Thus, as seen in the example in Fig. 2(a), while the edge $2\diamondsuit \rightarrow A\clubsuit$ is part of the cycle shown, if the edge $A\clubsuit \rightarrow 9\spadesuit$ is removed, which must be done before the edge $2\diamondsuit \rightarrow A\clubsuit$ can be removed, the cycle no longer exists. Thus, there is no reason to consider the edge $2\diamondsuit \rightarrow A\clubsuit$ as a candidate for removal during the calculation of edges required to be removed. We denote edges that must be considered as candidates for removal as *relevant cycle edges* and edges that need not be considered *irrelevant cycle edges*.

7 Duplicate Cycles

Since we are only concerned with the number of edges which must be removed to eliminate cycles, we can remove from consideration cycles which are duplicates of other cycles. We consider cycles to be duplicates if the set of rel-

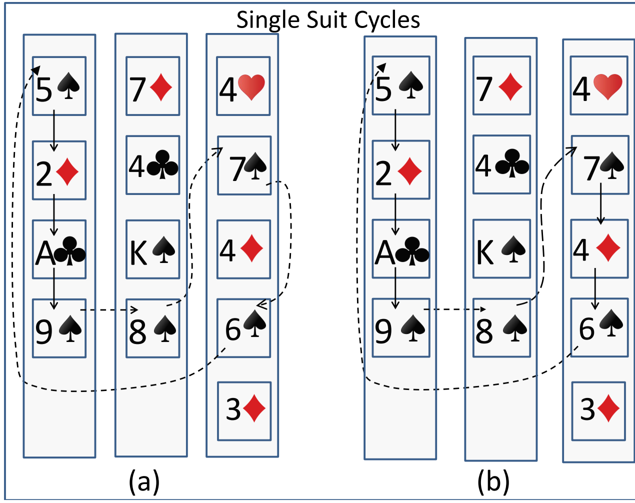


Figure 2: Fragment of tableau illustrating (a) single suit cycle in which all blocking edges are in a single column and (b) redundant single suit cycle consisting of blocking edges (in two columns) that are a superset of blocking edges in (a).

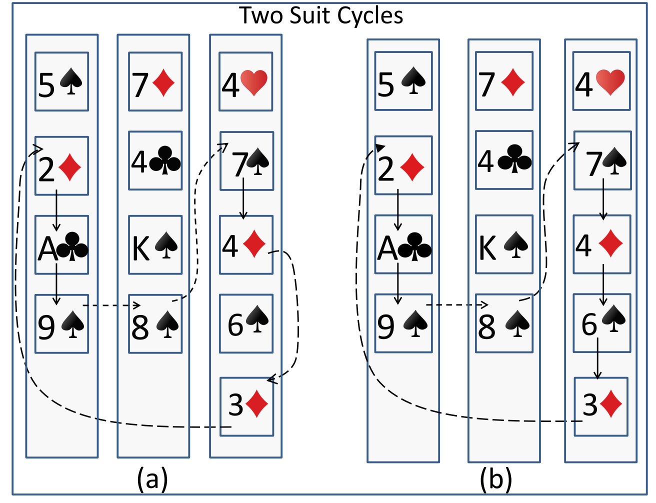


Figure 3: Fragment of tableau illustrating (a) 2-suit cycle in which all blocking edges are in a two columns and (b) redundant 2-suit cycle consisting of blocking edges (also in two columns) that are a superset of blocking edges in (a).

evant edges in the cycles are identical, independent of the order, and eliminate them from consideration.

8 Determination of the Minimum Number of Edges that Must be Removed to Eliminate Cycles

By eliminating redundant cycles, duplicate cycles and irrelevant edges of cycles, the complexity of determining the minimum number of edges that must be removed to eliminate cycles, m_e, can be reduced significantly. For example, we must actually only consider a total of 87 cycles (12 1-suit, 39 2-suit, 34 3-suit, and 2 4-suit cycles) for removal in the initial state of Freecell game $\#1$, compared to the 26133 cycles present if redundant and duplicate cycles are included.

We use a brute-force, depth-first exhaustive search of all combinations of edge removals that eliminate all cycles. Because only one edge of a cycle must be removed to eliminate the cycle, the worst case number of combinations of removed edges that we must consider is:

$$C = \prod_{i=1}^{m_c} e_i, \tag{3}$$

where m_c is the number of cycles and e_i is the number of relevant edges in cycle i.

In practice, the number of combinations actually considered can be reduced, in some cases by an order of magnitude, by removing edges in decreasing order of the number of other cycles in which an edge is contained. With this ordering many cycles are eliminated early in the calculation.

9 Implementation

Our solver is implemented in C++. In addition to eliminating redundant cycles, duplicate cycles and non-relevant edges, we improve performance with:

- A transposition table with Zobrist hashing to eliminate duplicate states during the A* search (Akagi, Kishimoto, and Fukunaga 2010; Zobrist 1990).
- The priority queue used by A* implemented as a series of buckets (as opposed to a binary heap), providing $O(1)$ queue performance (Paul 2007; Burns et al. 2012). This is possible because the costs (estimated solution lengths) are integers and in a relatively small range ($\approx$ 60–100).
- Use of a hash table to detect duplicate cycles.
- Incremental cycle determination. We identify all cycles in the initial configuration. After that, we incrementally identify cycles removed and added as a result of a move, only considering the card(s) moved.

10 Solver Results

Our test cases were games 1–5000 of Microsoft Freecell. We ran all tests on an Intel core i3 4160 processor running at 3.60 GHz with 8 GB of memory. The program working set is $\approx$ 2 GB for most games. Results for games 1–10 are shown in Table 1.

Before discussing the results, it is necessary to discuss the trade-off between the accuracy of our heuristic and the computer resources to achieve that accuracy. Our work can be thought of as providing a family of heuristic functions, $h_q(n)$, where $q = 1, 2, 3, 4$ is the maximum number of different suits of the dependency edges in the cycles we consider. Now, the processing time per state explored to determine cycles increases with q. The number of states which must be explored, however, decreases with q because the heuristic becomes more accurate with increasing q and the heuristic more effectively guides the search. Reducing the

Freecell Game #	initial state 1-suit cycles	2-suit cycles	length esti-mate	solution length	time (sec)	states searched
1	12	39	73	82	30.8	567699
2	13	5	68	73	1.9	101186
3	15	8	70	70	0.6	20499
4	37	34	72	79	31.0	1220026
5	16	46	78	85	122.0	3687136
6	12	24	73	75	1.1	31912
7	13	39	72	76	1.7	74369
8	8	58	70	74	13.2	367784
9	19	25	77	81	2.0	77990
10	16	18	73	80	7.7	315643

Table 1: Results of our program for the Freecell games 1-10.

number of states which must be explored is important because it also reduces the amount of memory A* requires and memory is often the limiting factor in A* implementations.

Varying q, we find that, while using cycles containing more than 2 suits reduces the number of states needed for a solution, the time per state is increased so much that the overall average time per solution is increased. For this reason, we did not identify or use cycles containing greater than 2 suits in our testing.

We found that across the 5000 games used for testing

- Optimal solutions have been found for all games tested.
- As expected, the initial estimated length is never greater than the actual optimal length – a requirement for the search to find the optimal solution.
- The initial estimated length and the actual optimal length are relatively close – in some cases (e.g., game 3) identical, indicating that the heuristic strongly guides the search.
- There are typically $\lessapprox 20$ initial 1-suit cycles and $\lessapprox 100$ 2-suit cycles.
- CPU processing time is dominated by the tasks of finding cycles and determining which edges must be removed to eliminate all cycles.
- The shortest, average and longest solution lengths were 64, 77 and 93 moves, respectively.
- The shortest, average and longest processing times were 0.4, 39.9 and 6579 seconds, respectively.

11 Solitaire and Blocks World

By mapping game states to directed graphs and using the number of edges required to remove cycles in the graph as input to the A* heuristic function, we develop, for the first time, an effective method for finding optimal solutions to skill-based solitaire games. These games are characterized by the presence of deadlock situations in which the goal state requires objects to be ordered in a specified way but constraints exist on the order in which actions can be taken. The presence of deadlocks in these games makes them hard to solve, and in particular hard to solve optimally.

This is reminiscent of the classic blocks world domain, where nonoptimal solutions can be generated easily, but computing optimal solutions is an NP-equivalent problem due to the existence of deadlocks (Gupta and Nau 1992; Slaney and Thiébaux 2001). Indeed, the Freecell heuristics described in this paper can be understood as a two-stage relaxation. Firstly, relax the Freecell game into a blocks world task. Secondly, compute an admissible heuristic for this blocks world task.

In more detail, we first relax the Freecell game by treating it as a blocks world task where the cards forming each tableau column or foundation pile are reinterpreted as a tower of blocks, and cards in free cells are reinterpreted as individual blocks lying on the table. This simplifies the problem (and hence leads to an admissible heuristic rather than a perfect distance estimate) because blocks world, unlike Freecell, has unlimited table positions. Interestingly, removing the constraint on table positions is the *only* relevant way in which this transformation simplifies the problem: while the Freecell rules impose a number of constraints regarding the movement of cards, none of these constraints affect the optimal solution length in the presence of unlimited table positions. In other words, Freecell with unlimited table positions (or unlimited free cells) always has exactly the same optimal solution length as the corresponding blocks world task.[4]

The second relaxation we apply in our experiments is to abstain from covering all deadlocks of the problem graph. Recall that h_q ($1 \leq q \leq 4$) is the variant of our heuristic that only considers q'-suit cycles with $q' \leq q$. The most

[4]To see this, observe that with unlimited space there is never an incentive in Freecell to move a card onto another card except for its final move to foundations. This is equivalent to the observation that in blocks world, there is never an incentive to move a block onto another block except to move it into its final position. The challenge, in both cases, is to minimize the number of moves of cards/blocks onto the table that cannot yet be moved directly into their final position.

powerful of these heuristics, h_4, includes all relevant cycles and hence amounts to solving the blocks world relaxation of the Freecell game perfectly. In our experiments, this level of heuristic accuracy turned out not to be beneficial due to the high computational effort for each state evaluation, with h_2 providing the best balance between heuristic accuracy and computational effort per state.

12 Implications for Domain-Independent Planning

Looking beyond solitaire games and blocks world, are there wider implications of our work for domain-independent planning? We believe that this is the case: that deadlocks are a phenomenon that occurs in a much wider range of domains than Freecell games or blocks world tasks, and that heuristic functions based on covering deadlocks are a promising direction for a wide range of planning domains.

For example, deadlocks of essentially the same form as in the blocks world domain are the major source of hardness in the Logistics domain and the only source of hardness in the Miconic-STRIPS and Miconic-SimpleADL domains (Helmert 2001). Many other planning domains with a "transportation" component share this problem aspect, though often mixed with other aspects. Deadlock covering problems also occur at the computational core of many optimization problems outside of planning, such as many of the *implicit hitting set* problems identified by Chandrasekaran et al. (2011).[5]

Finally, a similar form of deadlocks (a set of actions cyclically supporting each other's preconditions without being ultimately supported by effect/precondition links from the current state) is the major source of inaccuracy in *flow heuristics* that have recently attracted much attention in planning (van den Briel et al. 2007; Bonet 2013; Bonet and van den Briel 2014; Pommerening et al. 2014). A better understanding of the role of dependency deadlocks in classical planning tasks could go a long way towards overcoming the limitations of these heuristics.

References

Akagi, Y.; Kishimoto, A.; and Fukunaga, A. 2010. On transposition tables for single-agent search and planning: Summary of results. In Felner, A., and Sturtevant, N., eds., *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, 2–9. AAAI Press.

Bacchus, F. 2001. The AIPS'00 planning competition. *AI Magazine* 22(3):47–56.

Bonet, B., and van den Briel, M. 2014. Flow-based heuristics for optimal planning: Landmarks and merges. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 47–55. AAAI Press.

[5]See also the recent paper by Slaney (2014) for deeper connections between blocks world, implicit hitting sets, and combinatorial optimization in general.

Bonet, B. 2013. An admissible heuristic for SAS$^+$ planning obtained from the state equation. In Rossi, F., ed., *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2268–2274.

Burns, E.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing fast heuristic search code. In Borrajo, D.; Felner, A.; Korf, R.; Likhachev, M.; Linares López, C.; Ruml, W.; and Sturtevant, N., eds., *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012)*, 25–32. AAAI Press.

Chandrasekaran, K.; Karp, R.; Moreno-Centeno, E.; and Vempala, S. 2011. Algorithms for implicit hitting set problems. In Randall, D., ed., *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithm (SODA 2011)*, 614–629. SIAM.

Elyasaf, A.; Hauptman, A.; and Sipper, M. 2011. GA-FreeCell: evolving solvers for the game of FreeCell. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation (GECCO 2011)*, 1931–1938.

Elyasaf, A.; Hauptman, A.; and Sipper, M. 2012. Evolutionary design of FreeCell solvers. *IEEE Transactions on Computational Intelligence and AI in Games* 4(4):270–281.

Fish, S. 2015. Freecell solver. `http://fc-solve.shlomifish.org/`. Retrieved 11/9/2015.

Fox, M., and Long, D. 2001. Hybrid STAN: Identifying and managing combinatorial optimisation subproblems in planning. In Nebel, B., ed., *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, 445–452. Morgan Kaufmann.

FreeCell solutions. 2015. FreeCell solutions to 1000000 games. `http://freecellgamesolutions.com/`. Retrieved 11/9/2015.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Gupta, N., and Nau, D. S. 1992. On the complexity of blocks-world planning. *Artificial Intelligence* 56(2–3):223–254.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Heineman, G. T. 2015. Algorithm to solve FreeCell solitaire game. `http://broadcast.oreilly.com/2009/01/january-column-graph-algorithm.html`. Retrieved 11/9/2015.

Helmert, M. 2001. On the complexity of planning in transportation domains. In Cesta, A., and Borrajo, D., eds., *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, 120–126. AAAI Press.

Helmert, M. 2003. Complexity results for standard benchmark domains in planning. *Artificial Intelligence* 143(2):219–262.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Confer-*

ence on Automated Planning and Scheduling (ICAPS 2004), 161–170. AAAI Press.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research* 22:215–278.

Hoffmann, J. 2005. Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *Journal of Artificial Intelligence Research* 24:685–758.

Holte, R. C. 2010. Common misconceptions concerning heuristic search. In Felner, A., and Sturtevant, N., eds., *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, 46–51. AAAI Press.

Horne, J. 2015. Description of microsoft FreeCell shuffle algorithm. `http://www.solitairelaboratory.com/mshuffle.txt`. Retrieved 11/9/2015.

Keller, M. 2015. Solitaire laboratory. `http://solitairelaboratory.com/index.html`. Retrieved 11/9/2015.

Levin, J. 2008. Solitaire-y confinement: Why we can't stop playing a computerized card game. *Slate* May 16, 2008.

Long, D., and Fox, M. 2000. Automatic synthesis and use of generic types in planning. In Chien, S.; Kambhampati, S.; and Knoblock, C. A., eds., *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, 196–205. AAAI Press.

Mlot, S. 2015. Microsoft tournament celebrates 25 years of solitaire. `http://www.pcmag.com/article2/0,2817,2484370,00.asp`. Published May 19, 2015; retrieved 11/9/2015.

Morehead, A. H., and Mott-Smith, G. 1983. *The Complete Book of Solitaire and Patience Games*. Bantam.

Morris, R.; Tarassenko, L.; and Kenward, M. 2005. *Cognitive Systems – Information Processing Meets Brain Science*. Elsevier.

Paul, G. 2007. A complexity $o(1)$ priority queue for event driven molecular dynamics simulations. *Journal of Computational Physics* 221(2):615–625.

Pecora, F., and Cesta, A. 2003. The role of different solvers in planning and scheduling integration. In *Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence (AI*IA 2003)*, 362–374.

Pommerening, F.; Röger, G.; Helmert, M.; and Bonet, B. 2014. LP-based heuristics for cost-optimal planning. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 226–234. AAAI Press.

PySolFC. 2015. PySolFC: a Python solitaire game collection. `http://pysolfc.sourceforge.net/`. Retrieved 11/9/2015.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence — A Modern Approach*. Prentice Hall.

Sipper, M., and Elyasaf, A. 2014. Lunch isn't free, but cells are: Evolving FreeCell players. *SIGEvolution newsletter of the ACM Special Interest Group on Genetic and Evolutionary Computation* 6(3–4):2–10.

Slaney, J., and Thiébaux, S. 2001. Blocks World revisited. *Artificial Intelligence* 125(1–2):119–153.

Slaney, J. 2014. Set-theoretic duality: A fundamental feature of combinatorial optimisation. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., eds., *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, 843–848. IOS Press.

van den Briel, M.; Benton, J.; Kambhampati, S.; and Vossen, T. 2007. An LP-based heuristic for optimal planning. In Bessiere, C., ed., *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, 651–665. Springer-Verlag.

Van Noorden, R. 2006. Computer games could save your brain. *Nature* news item published 24 July 2006.

Wikipedia. 2015. FreeCell. `https://en.wikipedia.org/wiki/FreeCell`. Retrieved 11/9/2015.

Zobrist, A. L. 1990. A new hashing method with application for game playing. *ICCA Journal* 13(2):69–73.

A Freecell Rules

The following Freecell rules are taken from Wikipedia (2015).

Construction and layout: One standard 52-card deck is used. There are four open cells and four open foundations. Cards are dealt face-up into eight cascades, four of which comprise seven cards and four of which comprise six.

Building during play: The top card of each cascade begins a tableau. Tableaux must be built down by alternating colors. Foundations are built up by suit.

Moves: Any cell card or top card of any cascade may be moved to build on a tableau, or moved to an empty cell, an empty cascade, or its foundation. Complete or partial tableaus may be moved to build on existing tableaus, or moved to empty cascades, by recursively placing and removing cards through intermediate locations.

Victory: The game is won after all cards are moved to their foundation piles.

B Cycle Identification Algorithm

Figures 4 and 5 contain pseudo code for determining 1-suit and 2-suit cycles, respectively.

Note that there is no need to explicitly follow the dependency edges from a card in one column to a card of the same suit in another column since we are assured that there is always a chain of dependency edges from a card in a given suit to a card of lower rank in that suit.

Extension to 3-suit and 4-suit cycles is straightforward; paths of dependency edges to one or two additional columns, respectively, must be identified before returning to the initial column.

```
/* identify 1-suit cycles*/

for (all tableau columns,c)
{
   for (all cards, cardX,  in column c)
   {
      for (all cards,cardY, below cardX)
      {
         if (cardY suit != cardX suit || cardY rank > cardX rank)
            continue;   /* need same suit, lower rank than X*/

         /* cycle found */
         store cycle;
      }
   }
}
```

Figure 4: Pseudo code illustrating the algorithm to identify 1-suit cycles.

```
/* identify 2-suit cycles*/

for (all tableau columns, c1)
{
    for (all cards, cardX1, in column c1)
    {
        for (all tableau columns, c2)
        {
            for (all cards, cardY2 in column c2)
            {
                if (cardY2 suit != card X1 suit || cardY2 rank > cardX1 rank )
                    continue;    /* need same suit, lower rank than X1*/

                for (all cards, cardY1, above cardY2 in column c2)
                {
                    if (cardY1 suit == card Y2 suit)/* need different suit */
                        continue;

                    for (all cards, cardX2, below cardX1 in column c1)
                    {
                        if (cardX2 suit != cardY1 suit || cardX2 rank > cardY1 rank)
                            continue;    /* need same suit, lower rank than Y1*/

                        /* cycle found */
                        store cycle;
                    }
                }
            }
        }
    }
}
```

Figure 5: Pseudo code illustrating the algorithm to identify 2-suit cycles.

Lifting Delete Relaxation Heuristics To Successor Generator Planning

Michael Katz
IBM Watson Health, Israel
katzm@il.ibm.com

Dany Moshkovich
IBM Watson Health, Israel
mdany@il.ibm.com

Erez Karpas
Technion, Israel
karpase@technion.ac.il

Abstract

The problem of deterministic planning, i.e., of finding a sequence of actions leading from a given initial state to a goal, is one of the most basic and well studied problems in artificial intelligence. Two of the best known approaches to deterministic planning are the black box approach, in which a programmer implements a successor generator, and the model-based approach, in which a user describes the problem symbolically, e.g., in PDDL. While the black box approach is usually easier for programmers who are not experts in AI to understand, it does not scale up without informative heuristics. We propose an approach that we baptize as semi-black box (SBB) that combines the strength of both. SBB is implemented as a set of Java classes, which a programmer can inherit from when implementing a successor generator. Using the known characteristics of these classes, we can then automatically derive heuristics for the problem. Our empirical evaluation shows that these heuristics allow the planner to scale up significantly better than the traditional black box approach.

Introduction

The field of artificial intelligence has spent considerable effort on the seemingly simple problem of deterministic planning. At a high level, this problem can be formulated as: given an initial state, a desired goal, and a set of possible (deterministic) actions, find a sequence of actions which leads from the initial state to a state satisfying the goal. One popular approach to solving deterministic planning problems is heuristic search. However, two very different ways of using heuristic search algorithms to solve deterministic planning problems have been pursued throughout the history of the field.

The first approach, which we will refer to as the "black box" approach, involves implementing a piece of software to represent the planning problem. While the details of deterministic planning problems can be quite complex, it is enough to implement a very simple interface consisting of three functions: GET-INIT-STATE(), which returns an object representing the initial state, GET-SUCCESSORS(s), which returns the successors of a given state s, and IS-GOAL?(s), which checks whether the given state s is a goal state. Standard forward search algorithms, such as breadth first search, depth first search, or depth-first iterative deepening (Korf 1985), can use these three functions to solve the planning problem. However, in order to solve the problem more quickly, it is possible to use a *heuristic evaluation function*, or heuristic for short, which estimates the distance from a given state to the goal. Heuristic search algorithms such as A^* (Hart, Nilsson, and Raphael 1968) and its variants can use such a heuristic to solve the problems more quickly. Of course, the developer now also has to implement the H(s) function, in order to allow heuristic search algorithms to be used.

The second approach is the model-based approach (Geffner 2010), wherein one uses some symbolic language, such as PDDL (Mcdermott et al. 1998), to describe the planning problem. This typically involves defining a set of state variables and describing the initial state, the goal, and action preconditions and effects in terms of these state variables. It is then possible to automatically derive the same three functions mentioned above, as well as a heuristic evaluation function from the problem description (for example (Bonet and Geffner 1999)). Thus, it is possible to use the same heuristic search algorithms to solve domain-independent planning problems.

However, a major challenge for using the model-based approach to solve planning problems of interest to real-world users is that the average software developer has little to no experience with modeling. This is further compounded by the fact that some aspects of real world problems can be very hard to model symbolically, as evidenced by approaches such as planning with semantic attachments (Dornhege et al. 2009; Hertle et al. 2012) and planning modulu theories (Gregory et al. 2012), which allow the modeler to plug in external code in a general programming language to deal with specific aspects of the problem.

Our main motivation in this paper is the desire to make solving deterministic planning problems accessible to software developers who are not necessarily experts in artificial intelligence. The need to solve deterministic planning problems occurs not infrequently in real life, yet we are not aware of any frameworks which are both accessible to non-experts, and provide reasonable performance "out of the box".

In this paper, we describe such a framework, which brings the benefits of the model based approach, namely automatically derived heuristics, into black box successor generator planning. The key insight behind our framework is that, while planning problems can vary in their details, there are

some common underlying principles behind the vast majority of these problems. Our framework provides an implementation of these common principles, which is transparent to the model-based view, yet can still be used inside a "black box" implementation.

Background

We now describe the two approaches we mention above in more detail. We begin by defining a deterministic planning problem over a state space, which is a tuple $\Pi = \langle S, A, s_0, S_G, f \rangle$, where S is a finite set of *states*, A is a finite set of *action labels*, $s_0 \in S$ is the *initial state*, $S_G \subseteq S$ is the set of *goal states*, and $f : S \times A \to S$ is the *transition function*, such that $f(s, a)$ is the state which applying action a in state s leads to. A *solution* to such a problem is a sequence of action labels $\pi = \langle a_0, a_1, \dots a_n \rangle$, such that $f(f(f(s_0, a_0), a_1), \dots a_n) \in S_G$ — that is, a sequence of action labels which leads from the initial state to some goal state, using the transition function f.

While deterministic planning over a state space provides a nice mathematical model, the question of how the state space is described has more than one answer. The "black box" approach uses a tuple $\Pi_{bb} = \langle s_0, succ, goal? \rangle$, where s_0 is the initial state, $succ : S \to 2^{A \times S}$ is a successor generator, and $goal? : S \to \{T, F\}$ is the goal test function. In order to obtain a "black box" description of state space planning problem Π, we use the same initial state, and define $succ(s) = \{\langle a, s' \rangle \mid f(s, a) = s'\}$, and $goal?(s) = \begin{cases} T & s \in S_G \\ F & otherwise \end{cases}$.

On the other hand, the model-based approach assumes that the state space, S, can be *factored*, and represented by a set of variables. Different mathematical formalisms for such models exist (Fikes and Nilsson 1971; Bäckström and Nebel 1995), but we will focus on describing PDDL (Mcdermott et al. 1998), which includes both a mathematical formalism and a syntax for writing text files describing a planning problem in this formalism.

For ease of presentation, we describe a limited subset of PDDL, which corresponds to STRIPS (Fikes and Nilsson 1971). A planning task in PDDL is described by a tuple $\Pi_{pddl} = \langle O, P, Op, s_0, G \rangle$, where O is a set of objects, P is a set of predicates, Op is a set of operator schemas, s_0 is the initial state, and G is the goal condition. Each predicate $p \in P$ has an arity $ar(p)$, and defines the set of boolean propositions $\{p(o_1 \dots o_n) \mid ar(p) = n, o_1 \dots o_n \in O\}$. We will denote the union of these propositions from all predicates by F. Then the set of states defined by Π_{pddl} is 2^F, the initial state s_0 is defined by a list of the propositions which are true in the initial state, and the goal condition G is a list of propositions which we want to be true in the end, that is $S_G = \{s \mid G \subseteq s\}$.

Each operator scheme $op \in Op$ has a set of named arguments, $args(op)$, as well as a list of preconditions, add effects, and delete effects. Each element in these lists is a predicate $p \in P$, with a list of arguments from $args(op)$ of size $ar(p)$. A grounded action a is obtained from op by applying a substitution $\theta : args(op) \to O$ to all preconditions, add effects, and delete effects, which results in a 3-tuple $\langle pre(a), add(a), del(a) \rangle$, such that $pre(a), add(a), del(a) \subseteq F$. We can finally describe the transition function f that is defined by Π_{pddl}, as $f(s, a) = \begin{cases} (s \setminus del(a)) \cup add(a) & pre(a) \subseteq s \\ s & otherwise \end{cases}$.

Note that it is always possible to crate a PDDL description Π of a finite state space S, by defining a predicate of arity 0 for each state $s \in S$. However, the number of states that is described by this planning problem is exponential in $|S|$. Finding a *compact* PDDL description of a state space planning problem Π requires understanding the structure of Π, and is not always an easy task.

Additionally, PDDL is not always easy to deal with. For example, the occasional need to define actions with a very large number of parameters has been addressed by automatic domain transformations (Areces et al. 2014). PDDL also makes the "closed world assumption", that the only objects in the world are O. When this assumption does not hold, using PDDL planners is much more difficult (Talamadupula et al. 2010).

From Model-Based to Black Box

As previously mentioned, our objective is to provide developers who are not AI experts with off-the-shelf solvers to solve problems they are interested in. Modeling a problem in PDDL is often difficult for such non-experts. For example, many non-experts find it hard to understand why we can not define an action that moves agent A to location Y by $move(A, Y)$, and why we instead need to define the action as $move(A, X, Y)$. Thus, writing code to describe their problem (the "black box" approach) remains their only viable option. However, as the solver can not automatically derive a heuristic evaluation function using this approach, it is unlikely to scale.

In order to be able to combine both being able to programmatically specify the planning problem, and yet still be able to derive some heuristic guidance automatically, we propose a new framework, which we call *object oriented planning*. In this framework, the state of the planning problem is represented by a set of objects referred to as *entities*, each with their own internal state. The successor generator is defined by another set of classes, each of which represents a single operator, using two functions: IS-APPLICABLE(s,p)? which takes a state s and a list of parameters p, and checks if the action with parameters p is applicable in s, and APPLY(s,p) which returns the state resulting from applying the action with parameters p in state s. Note that while this is similar in spirit to PDDL, this is still a black box, since these functions are defined procedurally, not symbolically. The successor generator is implemented by calling IS-APPLICABLE? on the current state with all possible combinations of parameters, where each parameter can be any entity in the state. This allows the programmer to add and delete entities on the fly, a challenge with PDDL.

So far, we have described a framework which makes the "black box" approach slightly easier to use. However, the key idea behind our framework is that there is a small num-

ber of *stereotypes* of entities, which appear in many different planning domains. The developer can inherit from these stereotypes, saving some implementation effort. We also provide a number of *operator stereotypes* with known behavior. Since our framework understands these stereotypes, it can derive some heuristic guidance for these entities. The next section describes the stereotypes available in our prototype implementation.

Our current prototype implementation focuses on transportation domains, and supports two major stereotypes: *temporal*, which describes an entity with a clock, and *mobile*, which inherits the clock from *temporal* and can also be in one of several locations. Our framework also provides a *place* stereotype, which represents an immobile entity, and a *roadmap* interface, which allows the programmer to define the time and distance to travel directly between any two *places*, or to specify that they are not directly connected. For a mobile entity, we support specifying a set of temporally extended goal locations, constraining the allowed behavior of the entity.

Additionally, the framework provides *operator stereotypes*, which correspond to common operations on the entity stereotypes: *move*, which moves a mobile entity from one place to another, *load* which changes the location of a mobile entity to inside a mobile entity, and *unload*, which changes the location of a mobile entity from inside a mobile entity to the location of the entity. In the latter two cases, the clocks of all involved entities increase to reflect earliest applicability. As our framework is implemented in an object oriented language (specifically, in Java), the developer can inherit from the entity and action stereotypes, and implement the desired *additional* behavior, on top of what the framework provides.

Operators with the *move* stereotype take a *mobile* entity and a destination *place* as parameters, and update the location of the *mobile* entity to the destination, as well as incrementing the internal clock by the duration it takes to travel. Each such operator is associated with an instance of *roadmap* to use for obtaining the travel time. Thus, *walk* and *drive* can both inherit from *move*, be defined over the same set of *places*, but have different *roadmaps* defining different travel times, costs, and even connectivity. The cost of the operator is a linear combination of the travel time and travel distance, where the weights are specified by the programmer.

Operators with the *load* stereotype take two *mobile* entities, which must be in the same place, and change the location of the second to inside the first, and increment both their clocks to the maximum among their clocks plus the action duration. This represents having to meet up in the same place at the same time.

Operators with the *unload* stereotype take a *mobile* entity, which has been loaded inside another mobile entity, and unloads it, setting its location to the location of the external entity, and updates the clocks of both entities to the maximum among their clocks plus the action duration.

```
public class Vehicle extends MobileEntity {
  private int vehicleCurrentCapacity;
  private final int maximalCapacity;
  public Vehicle(String entityId, long time,
      long timeBound, Place location,
      RoutingRequest constraints,
      int maxCapacity) {
    super(entityId, time, timeBound,
                     location, constraints);
    maximalCapacity = maxCapacity;
    vehicleCurrentCapacity = -1;
  }
}

public class Participant
                      extends MobileEntity {
  private final String availableVehicleID;
  public Participant(String entityId,
               long time, long timeBound,
               Place location,
               RoutingRequest constraints,
               String vehicleID) {
    super(entityId, time, timeBound,
                     location, constraints);
    this.availableVehicleID = vehicleID;
  }
}
```

Figure 1: Commuter pooling domain Vehicle and Participant entities implementation example.

Examples

We now demonstrate the advantages of the Semi-Black Box approach on two concrete examples.

Commuter Pooling Domain

Our first example demonstrates the simplicity of modeling with the Semi-Black Box approach. We model a *commuter pooling* planning problem, where co-workers share rides on their way to work and back home.

A commuter pooling planning problem is defined by a set of participants P, which are *mobile* entities, as well as their home locations L and a work location w which are *places*. Some participants have a vehicle with limited capacity available, which is also a *mobile* entity. Figure 1 describes how these are implementated in our Semi-Black Box framework, and shows how Vehicle and Participant are implemented as classes which inherit from MobileEntity.

Each participant $p \in P$ is initially at her home location $home(p) \in L$, which she can leave no earlier than $hd(p)$, and arrive to the work location no later than $wa(p)$. On her way home, she can leave her work location no earlier than $wd(p)$, and arrive back home no later than $ha(p)$. These are implemented as temporally extended goals on p.

To model the *roadmap*, we implement the ILocationService interface, which specifies travel time and cost between different places. For each pair of locations, l_1, l_2, a duration and distance of moving from l_1 to l_2 are given by $\mathcal{T}(l_1, l_2)$ and $\mathcal{D}(l_1, l_2)$, respectively.

```
public class Drive extends Move {
  public Drive(
      ILocationService locationService){
    super("DRIVE_ACTION", Vehicle.class,
            Place.class, locationService,
            1, 0, 0);
  }

  @Override
  public boolean isApplicable(IState state,
          IEntity[] params) {
    Vehicle v = (Vehicle)params[0];
    return
      (v.getVehicleCurrentCapacity()>-1)
      && super.isApplicable(state,params);
  }
}
```

Figure 2: Implementation of Drive action in commuter pooling domain.

```
public class Board extends Load {

  public Board(){
    super("BOARD_ACTION", Participant.class,
          Vehicle.class, 1);
  }

  @Override
  public boolean isApplicable(IState state,
                     IEntity[] params){
    if (!super.isApplicable(state,params))
      return false;

    Participant p = (Participant)params[0];
    Vehicle v = (Vehicle) params[1];
    int capacity =
             v.getVehicleCurrentCapacity();

    if (capacity == -1)
      return p.getAvailableVehicleID()
                .equals(v.getEntityId());

    if (capacity < v.getMaximalCapacity())
      return !p.getAvailableVehicleID()
                .equals(v.getEntityId());

    return false;
  }

  @Override
  public void apply(IState state,
                         IEntity[] params){
    super.apply(state, params);

    Vehicle v = (Vehicle)params[1];
    int capacity =
           v.getVehicleCurrentCapacity();
    v.setVehicleCurrentCapacity(capacity+1);
  }
}
```

Figure 3: Implementation of Board action in commuter pooling domain.

Each participant's vehicle is initially located at that participant's home location. Each participant with an available vehicle can board/disembark that vehicle as a driver and any vehicle as a rider, as long as its full capacity is not reached. Vehicles with boarded drivers can drive between two connected locations. Figure 2 shows the implementation of the drive action, which inherits from Move. Note that the isApplicable method is overriden, and an extra check for checking if there is a driver in the car (v.getVehicleCurrentCapacity() > -1) is added.

Figure 3 shows the implementation of the board action, which checks if the vehicle is full or not by comparing current occupancy to the passenger capacity. In PDDL , this would have required having named slots for each seat. We omit the description of the other actions for the sake of brevity.

Evolution Domain

Our second example demonstrates that the Semi-Black Box approach is more expressive than PDDL. Our objective here is to create an organism that will merge the qualities of several organisms, a common task in evolutionary biology.

An Evolution planning task is defined by a set of organisms, who are either male or female. Each of them is initially at some location, and can move between locations. Two organisms of an opposite sex can reproduce, given that they are at the same location.

Given a subset of organisms G, the goal is to obtain a new organism, whose predecessors contain all organisms in G. Note that this planning problem involves creating an unknown number of new entities, and is therefore beyond the ability of PDDL to express.

Planning with Semi Black Box Representations

Having described our representation framework, we must now describe how we can solve problems formulated in this representation. We have already described how we can implement a successor generator and a goal test, and therefore we can use any uninformed search algorithm, such as BFS, DFS, ID-DFS (Korf 1985), *etc.* to solve the problem. However, uninformed search will not scale to large problem sizes.

In order to be able to scale up, we must make use of the extra information we have available — the model-based *portion* of the representation. Since we already have some known operator stereotypes, we exploit our knowledge of how these affect some aspects of the entities they are applied to, which also have known stereotypes. We do this by deriving a heuristic evaluation function, which estimates the distance from a given state to the goal. This allows us to use informed search algorithms, such as GBFS or weighted A* and solve larger problems.

Our framework provides operator stereotypes with known behavior, and entity stereotypes with known properties. Therefore, we derive a heuristic estimate of the distance to the goal by first "projecting" the problem onto its known aspects (that is, the known properties of entities and known

behavior of operators), and then deriving a heuristic estimate for this projection. Note that this projection is not a true abstraction in the formal sense of the word, as an operator with a known stereotype can modify its inherited behavior in arbitrary ways. However, that would constitute poor software engineering, and our purpose here is to provide a useful tool for software developers. Furthermore, even if the programmer did do this, it would only lead to inaccurate heuristic estimates, but will never affect the correctness of the plan that is returned.

We illustrate this point for our prototype implementation on mobile entities, and provide a PDDL-like description of this projection. The objects in our PDDL description are the set of entities and locations. The predicates we use are:

- Each mobile entity E can be in location L (at(E, L))
- Each mobile entity E can be inside another entity E' (in(E, E'))
- Each temporal entity (including mobile ones) has a clock with value T (time(E, T))
- For each mobile entity's temporally extended goal locations G, we need to indicate whether it was satisfied or not (satisfied(E, G)).

Finally, we can describe the effects of *move*, *load*, and *unload* using the above predicates.

While one might think it is possible to use any of the existing heuristics from the model-based planning community, there is a subtle issue here — unlike in PDDL, it is possible to add or delete entities on the fly in our framework. Therefore, if we ground the projection according to the initial state, as is commonly done in model-based planning, we might end up deriving a heuristic for the wrong problem as soon as some entity is added or deleted. Another issue is that with *temporal* entities, we can not ground their clocks, as the domain of the variable is all non-negative integers, and is thus unbounded. Therefore, we opt for computing a heuristic estimate over a lifted representation.

Devising meaningful estimates from lifted representation poses a challenge to the planning community (Ridder and Fox 2014). Our current implementation is a lifted variant of the $h_{\mathrm{FF}}(\Pi^C)$ heuristic (Keyder, Hoffmann, and Haslum 2014; Hoffmann and Fickert 2015). Given a set of sets of facts C, $h_{\mathrm{FF}}(\Pi^C)$ finds a semi-relaxed plan, in which delete effect interactions between the fluents in each set $X \in C$ are preserved. In our framework, these sets of fluents correspond to $\{\mathrm{AT}(E) \cup \mathrm{IN}(E) \cup \mathrm{TIME}(E) \cup \mathrm{SATISFIED}(E) \mid E$ is a moblie entity$\}$, where

- $\mathrm{AT}(E) = \{\mathrm{at}(E, L) \mid L$ is a location$\}$,
- $\mathrm{IN}(E) = \{\mathrm{in}(E, E') \mid E'$ is an entity$\}$,
- $\mathrm{TIME}(E) = \{\mathrm{time}(E, T) \mid T$ is a clock value$\}$, and
- $\mathrm{SATISFIED}(E) = \{\mathrm{satisfied}(E, G) \mid G$ is a temporally extended goal location$\}$.

Naturally, these sets are quite large, and impractical to be exploited in the grounded setting. In our framework, however, these sets correspond exactly to the possible values of mobile entities.

Specifically, we construct a variant of the relaxed planning graph, which is a layered graph, describing the relaxed action application from a given state. The layers are added until a fixpoint is reached, that is no new relaxed entity is added. During the construction of the graph, a successor generator is used to create concrete grounded instances of the *move*, *load*, and *unload* actions and add these instances to the graph. Additionally, if we can achieve some entity E being at some location L at two different times, we only keep the earliest. Thus, the overall procedure is guaranteed to terminate, since the aforementioned actions change mobile entities, with location having a finite number of possible values. Since each layer adds at least one such modified mobile entity to the graph, the overall bound on the number of layers is polynomial in the number of these values.

Once the relaxed planning graph is constructed, the last layer is checked to consists of a representative with all temporally extended goals on locations achieved for each mobile entity of the evaluated state. If that does not hold, an infinity value is returned. Otherwise, similarly to h_{FF}, the heuristic is computed using *best supporters* from either $h_{\max}$ or h_{add} heuristics on the nodes of the constructed layered graph[1] (Bonet and Geffner 2001; Keyder and Geffner 2008).

In order to speed up the heuristic computation, we introduced a simple dead end detection check, validating that each mobile entity can reach each location it is explicitly constrained to visit within the defined temporal bounds in a relaxed fashion. We note that our implementation of the heuristic function is rather naive and can be significantly sped up by introducing sophisticated data structures, etc.

Related work

We are not the first to identify the difficulty of using symbolic languages such as PDDL to model some interesting, useful planning problems. Functional STRIPS (Geffner 2000; Francés and Geffner 2015) introduces function symbols which can be nested, and thus allows us to have objects without explicit names — something that PDDL does not support.

Planning with semantic attachments (Dornhege et al. 2009; Hertle et al. 2012) and planning modulu theories (Gregory et al. 2012) both allow the user to combine symbolic models with more expressive modules (or theories), which are implemented as external function calls to a generic programming language. These external calls are tied into the symbolic model via an interface involving a set of predicates of the symbolic model.

None of the approaches described above alleviate the need for symbolic modeling. In fact, they force the user to think of a good abstraction for the external modules, which will serve as the interface. Our approach, on the other hand, frees the user from the need for symbolic modelling, except where she specifically chooses to do so.

[1]We currently do not implement the preferred operators feature that proved to be extremely helpful in the model-based planning, leaving it for the future work.

	Plan cost				Quality				Total time				
	LAMA	FF	SBB	BB	LAMA	FF	SBB	BB	optic	LAMA	FF	SBB	BB
02_0	1108	1108	1108	1108	1.00	1.00	1.00	1.00	0.1	0.2	0.1	0.1	0.1
02_1	1108	1108	1108	1108	1.00	1.00	1.00	1.00	0.0	0.1	0.1	0.1	0.0
04_0	2176		1136		0.52	0.00	1.00	0.00	8.3	1322.5		280.4	
04_1	1136	1136	1136		1.00	1.00	1.00	0.00	0.3	1203.7	737.8	29.3	
04_2	1140	1140	1140	1140	1.00	1.00	1.00	1.00	22.2	93.2	25.9	1.5	2.0
04_3	1136	1136	1136	1136	1.00	1.00	1.00	1.00	0.2	4.8	1.1	0.3	0.6
06_0	5324		4332		0.81	0.00	1.00	0.00		50.3		3.5	
06_1	5364		5374		1.00	0.00	1.00	0.00		22.2		5.3	
06_2	4304		3270		0.76	0.00	1.00	0.00	437.4	35.1		1.1	
06_3	3264	3304	2244		0.69	0.68	1.00	0.00		22.6	627.0	471.7	
06_4	2244	2244	2244	2244	1.00	1.00	1.00	1.00		54.7	304.6	6.7	25.9
08_0	7472		5426		0.73	0.00	1.00	0.00		151.3		39.0	
08_1	6412		6402		1.00	0.00	1.00	0.00		208.6		6.7	
08_6			∞		0.00	0.00	1.00	0.00				239.9	
10_0	9560				1.00	0.00	0.00	0.00		244.6			
10_2	8560				1.00	0.00	0.00	0.00		368.4			
Sum					13.51	6.68	14.00	5.00					

Table 1: Empirical Results on Commuter Pooling Domain.

task	03_3	04_3	04_4	06_3	06_4	08_3	08_4	10_3
cost	250	250	340	240	320	220	320	220
time	0.564	0.311	0.511	1.27	1.224	0.482	6.88	2.368

Table 2: Empirical Results for SBB on Evolution Domain.

Empirical evaluation

In order to empirically evaluate the effectiveness of solving complex problems with the semi-black box approach, we implemented the approach in Java, together with the greedy best-first search, and the lazy weighted A^* search. The comparison was performed on 25 generated problems of an increasing size of the commuter pooling domain. The results are depicted in Table 1, showing the instances where at least one of the planners was able to find a solution. We used a 2GB memory bound and 30 minutes time bound on a single core of an Intel(R) Core(TM) i7 2.5 GHz machine.

Our approach (SBB in Table 1) performs an iterative search with found solution cost passed as an upper bound to the next iteration, similarly to the LAMA planner (Richter and Westphal 2010). We start with a greedy best first search, and then weighted A^* with decreasing weights 5, 3, 2, and 1, continuing with weight 1 until no solution is found. First, we compare our approach to a state-of-the-art temporal planner *optic* (Benton, Coles, and Coles 2012). Second, we compare to the pure black box approach (BB in Table 1) — BFS without the automatically derived heuristic.

The commuter pooling domain corresponds to a *temporally simple* fragment of temporal planning, and thus can be mapped to STRIPS in linear time (Cushing et al. 2007). Therefore, we also compare to two classical planners, manually adjusting the time granularity and manually removing (unrecognized by the preprocessor) unreachable time values, to allow for successful grounding of reasonable size tasks. We used the Fast Downward planning framework (Helmert 2006) with two configurations: an iterative search with the FF heuristic (Hoffmann and Nebel 2001) without preferred operators, which is the closest configuration to our solution method (FF in Table 1), and the state-of-the-art LAMA planner (Richter and Westphal 2010).

The leftmost part of Table 1 shows the best found plan cost for four of the approaches that aim at optimizing plan cost. The middle part shows the best obtained solution quality, which is a standard IPC score allocating a number between 0 and 1 to each run, where 1 is given to a planner that found the best solution for that task, and 0 stands for not being able to solve the task within the given bounds. The rightmost part shows the total run time until the best solution was found.

As these results show, SBB outperforms all other planners on IPC score. Comparing to the second best performer, LAMA, LAMA solves two instances that SBB did not, while SBB solves one instance that LAMA did not (proving that it is infeasible). On instances that they both solve, SBB is typically much faster, except for a single instance. A comparison to the most similar technique, FF, shows that SBB is much better, indicating that there is some value in the lifted heuristic computation. Finally, comparing to BB shows the automatically derived heuristic is essential.

In addition, to test the feasibility of our approach for solving tasks outside the PDDL fragment, we performed an evaluation of the Evolution domain. The results are depicted in Table 2. The tasks are named x_y, where x is the number of initially existing organizms, and y is the size of G, the subset of organizms that should be among the predecessors of the target organizm. The results clearly show that our approach is able to cope with sufficiently large instances.

Discussion and future work

We introduce a framework that brings the benefits of the model based approach into black box successor generator planning by allowing annotating planning problem entities and actions with certain predefined stereotypes. By that, we take a major step toward making solving deterministic plan-

ning problems accessible to software developers who are not necessarily experts in artificial intelligence.

For future work, we intend to extend our framework by both introducing and exploiting additional stereotypes, and by introducing additional search enhancements, such as additional automatically derived heuristics (landmarks, abstractions) and search boosting techniques, such as preferred operators.

References

Areces, C.; Bustos, F.; Dominguez, M. A.; and Hoffmann, J. 2014. Optimizing planning domains by automatic action schema splitting. In *Proc. ICAPS 2014*.

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11:625–656.

Benton, J.; Coles, A. J.; and Coles, A. 2012. Temporal planning with preferences and time-dependent continuous costs. In *Proc. ICAPS 2012*.

Bonet, B., and Geffner, H. 1999. Planning as heuristic search: New results. In *Proc. ECP 1999*, 360–372.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. S. 2007. When is temporal planning really temporal? In *Proc. IJCAI 2007*, 1852–1859.

Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009. Semantic attachments for domain-independent planning systems. In *Proc. ICAPS 2009*.

Fikes, R., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3/4):189–208.

Francés, G., and Geffner, H. 2015. Modeling and computation in planning: Better heuristics from more expressive languages. In *Proc. ICAPS 2015*.

Geffner, H. 2000. Functional strips: A more flexible language for planning and problem solving. In Minker, J., ed., *Logic-Based Artificial Intelligence*, volume 597 of *The Springer International Series in Engineering and Computer Science*. Springer US. 187–209.

Geffner, H. 2010. The model-based approach to autonomous behavior: A personal view. In *Proc. AAAI 2010*.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning modulo theories: Extending the planning paradigm. In *Proc. ICAPS 2012*.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics* 4(2):100–107.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Intell. Res. (JAIR)* 26:191–246.

Hertle, A.; Dornhege, C.; Keller, T.; and Nebel, B. 2012. Planning with semantic attachments: An object-oriented view. In *Proc. ECAI 2012*, 402–407.

Hoffmann, J., and Fickert, M. 2015. Explicit conjunctions without compilation: Computing $h^{ff}(pi^{c})$ in polynomial time. In *Proc. ICAPS 2015*, 115–119.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proc. ECAI 2008*, 588–592.

Keyder, E. R.; Hoffmann, J.; and Haslum, P. 2014. Improving delete relaxation heuristics through explicitly represented conjunctions. *J. Artif. Intell. Res. (JAIR)* 50:487–533.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27:97–109.

Mcdermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – the planning domain definition language. Technical Report TR-98-003, Yale Center for Computational Vision and Control.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res. (JAIR)* 39:127–177.

Ridder, B., and Fox, M. 2014. Heuristic evaluation based on lifted relaxed planning graphs. In *Proc. ICAPS 2014*.

Talamadupula, K.; Benton, J.; Schermerhorn, P. W.; Kambhampati, S.; and Scheutz, M. 2010. Integrating a closed world planner with an open world robot: A case study. In *Proc. AAAI 2010*.

Non-Deterministic Planning with Temporally Extended Goals: Completing the story for finite and infinite LTL

Alberto Camacho[†], Eleni Triantafillou[†], Christian Muise[*], Jorge A. Baier[‡], Sheila A. McIlraith[†]
[†]Department of Computer Science, University of Toronto
[*]CSAIL, Massachusetts Institute of Technology
[‡]Departamento de Ciencia de la Computación, Pontificia Universidad Católica de Chile
[†]{acamacho,eleni,sheila}@cs.toronto.edu, [*]{cjmuise}@mit.edu, [‡]{jabaier}@ing.puc.cl

Abstract

Temporally extended goals are critical to the specification of a diversity of real-world planning problems. Here we examine the problem of planning with temporally extended goals over both finite and infinite traces where actions can be non-deterministic, and where temporally extended goals are specified in linear temporal logic (LTL). Unlike existing LTL planners, we place no restrictions on our LTL formulae beyond those necessary to distinguish finite from infinite trace interpretations. We realize our planner by compiling temporally extended goals, represented in LTL, into Planning Domain Definition Language problem instances, and exploiting a state-of-the-art fully observable non-deterministic planner to compute solutions. The resulting planner is sound and complete. Our approach exploits the correspondence between LTL and automata. We propose several different compilations based on translations of LTL to (Büchi) alternating or non-deterministic finite state automata, and evaluate various properties of the competing approaches. We address a diverse spectrum of LTL planning problems that, to this point, had not been solvable using AI planning techniques. We do so while demonstrating competitive performance relative to the state of the art in LTL planning.

1 Introduction

Most real-world planning problems involve complex goals that are temporally extended, require adherence to safety constraints and directives, necessitate the optimization of preferences or other quality measures, and/or require or may benefit from following a prescribed high-level script that specifies *how* the task is to be realized. In this paper we focus on the problem of planning for temporally extended goals, constraints, directives or scripts that are expressed in Linear Temporal Logic (LTL) for planning domains in which actions can have *non-deterministic* effects, and where LTL is interpreted over either finite or infinite traces.

Planning with deterministic actions and LTL goals has been well studied, commencing with the works of Bacchus and Kabanza (2000) and Doherty and Kvarnström (2001). Significant attention has been given to compilation-based approaches (e.g., (Rintanen 2000; Cresswell and Coddington 2004; Edelkamp 2006; Baier and McIlraith 2006; Patrizi et al. 2011)), which take a planning problem with an LTL goal and transform it into a classical planning problem for which state-of-the-art classical planning technology can often be leveraged. The more challenging problem of planning with non-deterministic actions and LTL goals has not been studied to the same extent; Kabanza, Barbeau, and St.-Denis (1997), and Pistore and Traverso (2001) have proposed their own LTL planners, while Patrizi, Lipovetzky, and Geffner (2013) have proposed the only compilation-based approach that exists. Unfortunately, the latter approach is limited to the proper subset of LTL for which there exists a *deterministic* Büchi automata. In addition, it is restricted to the interpretation of LTL over *infinite* traces and the compilation is worst-case exponential in the size of the goal formula.

In this paper, we propose a number of compilation-based approaches for LTL planning with non-deterministic actions. Specifically, we present two approaches for LTL planning with non-deterministic actions over infinite traces and two approaches for LTL planning with non-deterministic actions over finite traces[1]. In each case, we exploit translations from LTL to (Büchi) alternating or non-deterministic finite state automata. All of our compilations are sound and complete and result in Planning Domain Definition Language (PDDL) encodings suitable for input to standard fully observable non-deterministic (FOND) planners. Our compilations based on alternating automata are linear in time and space with respect to the size of the LTL formula, while those based on non-deterministic finite state automata are worst-case exponential in time and space (although optimizations in the implementation avoid this in our experimental analysis).

Our approaches build on methods for finite LTL planning with deterministic actions by Baier and McIlraith (2006) and Torres and Baier (2015), and for the infinite non-deterministic case, on the work of Patrizi, Lipovetzky, and Geffner (2013). While in the finite case the adaptation of these methods was reasonably straightforward, the infinite case required non-trivial insights and modifications to Torres and Baier's approach. We evaluate the relative performance of our compilation-based approaches using state-of-the-art FOND planner PRP (Muise, McIlraith, and Beck 2012), demonstrating that they are competitive with state-of-the-art LTL planning techniques.

[1]Subtleties relating to the interpretation of LTL over finite traces are discussed in (De Giacomo and Vardi 2013).

Our work presents the first realization of a compilation-based approach to planning with *non-deterministic* actions where the LTL is interpreted over finite traces. Furthermore, unlike previous approaches to LTL planning, our compilations make it possible, for the first time, to solve the complete spectrum of FOND planning with LTL goals interpreted over infinite traces. Indeed, all of our translations capture the full expressivity of the LTL language. Table 1 summarizes existing compilation-based approaches and the contributions of this work. Our compilations enable a diversity of real-world planning problems as well as supporting a number of applications outside planning proper ranging from business process analysis, and web service composition to narrative generation, automated diagnosis, and automated verification. Finally and importantly, our compilations can be seen as a practical step towards the efficient realization of a class of LTL synthesis tasks using planning technology (e.g., (Pnueli and Rosner 1989; De Giacomo and Vardi 2015)). We elaborate further with respect to related work in Section 5.

2 Preliminaries

2.1 FOND Planning

Following Ghallab, Nau, and Traverso (2004), a *Fully Observable Non-Deterministic* (FOND) planning problem consists of a tuple $\langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$, where $\mathcal{F}$ is a set of propositions that we call *fluents*, $\mathcal{I} \subseteq \mathcal{F}$ characterizes what holds in the initial state; $\mathcal{G} \subseteq \mathcal{F}$ characterizes what must hold for the goal to be achieved. Finally $\mathcal{A}$ is the set of actions. The set of literals of $\mathcal{F}$ is $Lits(\mathcal{F}) = \mathcal{F} \cup \{\neg f \mid f \in \mathcal{F}\}$. Each action $a \in \mathcal{A}$ is associated with $\langle Pre_a, Eff_a \rangle$, where $Pre_a \subseteq Lits(\mathcal{F})$ is the precondition and Eff_a is a set of outcomes of a. Each outcome $e \in Eff_a$ is a set of conditional effects, each of the form $(C \rightarrow \ell)$, where $C \subseteq Lits(\mathcal{F})$ and $\ell \in Lits(\mathcal{F})$. Given a planning state $s \subseteq \mathcal{F}$ and a fluent $f \in \mathcal{F}$, we say that s satisfies f, denoted $s \models f$ iff $f \in s$. In addition $s \models \neg f$ if $f \notin s$, and $s \models L$ for a set of literals L, if $s \models \ell$ for every $\ell \in L$. Action a is *applicable* in state s if $s \models Pre_a$. We say s' *is a result of applying a in s* iff, for some e in Eff_a, s' is equal to $s \setminus \{p \mid (C \rightarrow \neg p) \in e, s \models C\} \cup \{p \mid (C \rightarrow p) \in e, s \models C\}$. The *determinization* of a FOND problem $\langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ is the planning problem $\langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A}' \rangle$, where each non-deterministic action $a \in \mathcal{A}$ is replaced by a set of deterministic actions, a_i, one action corresponding to each of the distinct non-deterministic effects of a. Together these deterministic actions comprise the set $\mathcal{A}'$.

Solutions to a FOND planning problem $\mathcal{P}$ are *policies*. A policy p is a partial function from states to actions such that if $p(s) = a$, then a is applicable in s. The *execution* of a policy p in state s is an infinite sequence $s_0, a_0, s_1, a_1, \ldots$ or a finite sequence $s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n$, where $s_0 = s$, and all of its state-action-state substrings s, a, s' are such that $p(s) = a$ and s' is a result of applying a in s. Finite executions ending in a state s are such that $p(s)$ is undefined. An execution σ *yields* the state trace π that results from removing all the action symbols from σ.

Alternatively, solutions to $\mathcal{P}$ can be represented by means of *finite-state controllers* (FSCs). Formally, a FSC is a tuple $\Pi = \langle C, c_0, \Gamma, \Lambda, \rho, \Omega \rangle$, where C is the set of *controller states*, $c_0 \in C$ is the *initial controller state*, $\Gamma = S$ is the *input alphabet* of Π, $\Lambda = \mathcal{A}$ is the *output alphabet* of Π, $\rho : C \times \Gamma \rightarrow C$ is the *transition function*, and $\Omega : C \rightarrow \Lambda$ is the controller *output function* (cf. (Geffner and Bonet 2013; Patrizi, Lipovetzky, and Geffner 2013)). In a planning state s, Π outputs action $\Omega(c_i)$ when the controller state is c_i. Then, the controller transitions to state $c_{i+1} = \rho(c_i, s')$ if s' is the new planning state, assumed to be fully observable, that results from applying $\Omega(c_i)$ in s. The *execution* of a FSC Π in controller state c (assumed to be $c = c_0$) and state s is an infinite sequence $s_0, a_0, s_1, a_1, \ldots$ or a finite sequence $s_0, a_0, \ldots, s_{n-1}, a_{n-1}, s_n$, where $s_0 = s$, and such that all of its state-action-state substrings s_i, a_i, s_{i+1} are such that $\Omega(c_i) = a_i$, s_{i+1} is a result of applying a_i in s_i, and $c_{i+1} = \rho(c_i, s_i)$. Finite executions ending in a state s_n are such that $\Omega(c_n)$ is undefined. An execution σ *yields* the state trace π that results from removing all the action symbols from σ.

Following Geffner and Bonet (2013), an infinite execution σ is *fair* iff whenever s, a occurs infinitely often within σ, then so does s, a, s', for every s' that is a result of applying a in s. A solution is a *strong cyclic plan* for $\langle \mathcal{F}, \mathcal{I}, \mathcal{G}, \mathcal{A} \rangle$ iff each of its executions in $\mathcal{I}$ is either finite and ends in a state that satisfies $\mathcal{G}$ or is (infinite and) unfair.

2.2 Linear Temporal Logic

Linear Temporal Logic (LTL) was first proposed for verification (Pnueli 1977). An LTL formula is interpreted over an infinite sequence, or *trace*, of states. Because the execution of a sequence of actions induces a trace of planning states, LTL can be naturally used to specify temporally extended planning goals when the execution of the plan naturally yields an infinite state trace, as may be the case in non-deterministic planning.

In classical planning –i.e. planning with deterministic actions and final-state goals–, plans are finite sequences of actions which yield finite execution traces. As such, approaches to planning with deterministic actions and LTL goals (e.g., (Baier and McIlraith 2006)), including the Planning Domain Definition Language (PDDL) version 3 (Gerevini and Long 2005), use a *finite* semantics for LTL, whereby the goal formula is evaluated over a finite state trace. De Giacomo and Vardi (2013) formally described and analyzed such a version of LTL, which they called $\mathsf{LTL_f}$, noting the distinction with LTL (De Giacomo, Masellis, and Montali 2014).

LTL and $\mathsf{LTL_f}$ allow the use of modal operators *next* ($\bigcirc$), and *until* (U), from which it is possible to define the well-known operators *always* ($\Box$) and *eventually* ($\Diamond$). $\mathsf{LTL_f}$, in addition, allows a *weak next* ($\bullet$) operator. An $\mathsf{LTL_f}$ formula over a set of propositions $\mathcal{P}$ is defined inductively: a proposition in $\mathcal{P}$ is a formula, and if ψ and χ are formulae, then so are $\neg\psi$, $(\psi \wedge \chi)$, $(\psi \mathsf{U} \chi)$, $\bigcirc\psi$, and $\bullet\psi$. LTL is defined analogously.

The semantics of LTL and $\mathsf{LTL_f}$ is defined as follows. Formally, a state trace π is a sequence of states, where each state is an element in $2^{\mathcal{P}}$. We assume that the first state in π

Infinite LTL		Finite LTL	
Deterministic Actions	Non-Deterministic Actions	Deterministic Actions	Non-Deterministic Actions
[Albarghouthi et al., 2009] (EXP) [Patrizi et al., 2011] (EXP)	[Patrizi et al., 2013] (limited LTL) (EXP) **[this paper (BAA)]** (LIN) **[this paper (NBA)]** (EXP)	[Edelkamp, 2006] (EXP) [Cresswell & Coddington, 2006] (EXP) [Baier & McIlraith, 2006] (EXP) [Torres & Baier, 2015] (LIN)	**[this paper (NFA)]** (EXP) **[this paper (AA)]** (LIN)

Table 1: Automata-based compilation approaches for LTL planning. (EXP): worst case exponential. (LIN): linear.

is s_1, that the i-th state of π is s_i and that $|\pi|$ is the length of π (which is ∞ if π is infinite). We say that π satisfies φ ($\pi \models \varphi$, for short) iff $\pi, 1 \models \varphi$, where for every natural number $i \geq 1$:

- $\pi, i \models p$, for a propositional variable $p \in \mathcal{P}$, iff $p \in s_i$,
- $\pi, i \models \neg\psi$ iff it is not the case that $\pi, i \models \psi$,
- $\pi, i \models (\psi \wedge \chi)$ iff $\pi, i \models \psi$ and $\pi, i \models \chi$,
- $\pi, i \models \bigcirc\varphi$ iff $i < |\pi|$ and $\pi, i+1 \models \varphi$,
- $\pi, i \models (\varphi_1 \mathsf{U} \varphi_2)$ iff for some j in $\{i, \ldots, |\pi|\}$, it holds that $\pi, j \models \varphi_2$ and for all $k \in \{i, \ldots, j-1\}$, $\pi, k \models \varphi_1$,
- $\pi, i \models \bullet\varphi$ iff $i = |\pi|$ or $\pi, i+1 \models \varphi$.

Observe operator $\bullet$ is equivalent to $\bigcirc$ iff π is infinite. Therefore, henceforth we allow $\bullet$ in LTL formulae, we do not use the acronym $\mathsf{LTL_f}$, but we are explicit regarding which interpretation we use (either finite or infinite) when not obvious from the context. As usual, $\Diamond\varphi$ is defined as $(\mathsf{true} \, \mathsf{U} \, \varphi)$, and $\Box\varphi$ as $\neg\Diamond\neg\varphi$. We use the *release* operator, defined by $(\psi \, \mathsf{R} \, \chi) \stackrel{\text{def}}{=} \neg(\neg\psi \, \mathsf{U} \, \neg\chi)$.

2.3 LTL, Automata, and Planning

Regardless of whether the interpretation is over an infinite or finite trace, given an LTL formula φ there exists an automata $\mathcal{A}_\varphi$ that accepts a trace π iff $\pi \models \varphi$. For infinite interpretations of φ, a trace π is accepting when the run of (a Büchi non-deterministic automata) $\mathcal{A}_\varphi$ on π visits its accepting states infinitely often. For finite interpretations, π is accepting when the final automata state is accepting. For the infinite case such automata may be either Büchi non-deterministic or Büchi alternating (Vardi and Wolper 1994), whereas for the finite case such automata may be either non deterministic (Baier and McIlraith 2006) or alternating (De Giacomo, Masellis, and Montali 2014; Torres and Baier 2015). Alternation allows the generation of compact automata; specifically, $\mathcal{A}_\varphi$ is linear in the size of φ (both in the infinite and finite case), whereas the size of non-deterministic (Büchi) automata is worst-case exponential.

These automata constructions have been exploited in deterministic and non-deterministic planning with LTL via compilation approaches that allow us to use existing planning technology for non-temporal goals. The different state of the art automata-based approaches for deterministic and FOND LTL planning are summarized in Table 1. Patrizi, Lipovetzky, and Geffner (2013) present a Büchi automata-based compilation for that subset of LTL which relies on the construction of a Büchi *deterministic* automata. It is a well-known fact that Büchi deterministic automata are not equivalent to Büchi non-deterministic automata, and thus this last approach is applicable to a limited subset of LTL formulae.

3 FOND Planning with LTL Goals

An LTL-FOND planning problem is a tuple $\langle \mathcal{F}, \mathcal{I}, \varphi, \mathcal{A} \rangle$, where $\mathcal{F}$, $\mathcal{I}$, and $\mathcal{A}$ are defined as in FOND problems, and φ is an LTL formula. Solutions to an LTL-FOND problem are FSCs, as described below.

Definition 1 (Finite LTL-FOND)**.** *An FSC Π is a solution for $\langle \mathcal{F}, \mathcal{I}, \varphi, \mathcal{A} \rangle$ under the finite semantics iff every execution of Π over $\mathcal{I}$ is such that either (1) it is finite and yields a state trace π such that $\pi \models \varphi$ or (2) it is (infinite and) unfair.*

Definition 2 (Infinite LTL-FOND)**.** *An FSC Π is a solution for $\langle \mathcal{F}, \mathcal{I}, \varphi, \mathcal{A} \rangle$ under the infinite semantics iff (1) every execution of Π over $\mathcal{I}$ is infinite and (2) every fair (infinite) execution yields a state trace π such that $\pi \models \varphi$.*

Below we present two general approaches to solving LTL-FOND planning problems by compiling them into standard FOND problems. Each exploits correspondences between LTL and either alternating or non-deterministic automata, and each is specialized, as necessary, to deal with LTL interpreted over either infinite (Section 3.1) or finite (Section 3.2) traces. We show that FSC representations of strong-cyclic solutions to the resultant FOND problem are solutions to the original LTL-FOND problem. Our approaches are the first to address the full spectrum of FOND planning with LTL interpreted over finite and inifinte traces. In particular our work is the first to solve *full* LTL-FOND with respect to infinite trace interpretations, and represents the first realization of a compilation approach for LTL-FOND with respect to finite trace interpretations.

3.1 From Infinite LTL-FOND to FOND

We present two different approaches to infinite LTL-FOND planning. The first approach exploits Büchi alternating automata (BAA) and is linear in time and space with respect to the size of the LTL formula. The second approach exploits Büchi non-deterministic automata (NBA), and is worst-case exponential in time and space with respect to the size of the LTL formula. Nevertheless, as we see in Section 4, the second compilation does not exhibit this worst-case complexity in practice, generating high quality solutions with reduced compilation run times and competitive search performance.

3.1.1 A BAA-based Compilation Our BAA-based compilation builds on ideas by Torres and Baier (2015) for alternating automata (AA) based compilation of *finite* LTL planning with *deterministic* actions (henceforth, TB15), and from Patrizi, Lipovetzky, and Geffner's compilation (2013)

(henceforth, PLG13) of LTL-FOND to FOND. Combining these two approaches is not straightforward. Among other reasons, TB15 does not yield a sound translation for the infinite case, and thus we needed to modify it significantly. This is because the accepting condition for BAAs is more involved than that of regular AAs.

The first step in the compilation is to build a BAA for our LTL goal formula φ over propositions $\mathcal{F}$, which we henceforth assume to be in negation normal form (NNF). Transforming an LTL formula φ to NNF can be done in linear time in the size of φ. The BAA we use below is an adaptation of the BAA by Vardi (1995). Formally, it is represented by a tuple $\mathcal{A}_\varphi = (Q, \Sigma, \delta, q_\varphi, Q_{Fin})$, where the set of states, Q, is the set of subformulae of φ, $sub(\varphi)$ (including φ), Σ contains all sets of propositions in $\mathcal{P}$, $Q_{Fin} = \{\alpha \,\mathsf{R}\, \beta \in sub(\varphi)\}$, and the transition function, δ is given by:

$$\delta(\ell, s) = \begin{cases} \top & \text{if } s \models \ell \ (\ell, \text{literal}) \\ \bot & \text{otherwise} \end{cases}$$
$$\delta(\alpha \wedge \beta, s) = \delta(\alpha, s) \wedge \delta(\beta, s)$$
$$\delta(\bigcirc\alpha, s) = \alpha$$
$$\delta(\alpha \vee \beta, s) = \delta(\alpha, s) \vee \delta(\beta, s)$$
$$\delta(\alpha \,\mathsf{U}\, \beta, s) = \delta(\beta, s) \vee (\delta(\alpha, s) \wedge \alpha \,\mathsf{U}\, \beta)$$
$$\delta(\alpha \,\mathsf{R}\, \beta, s) = \delta(\beta, s) \wedge (\delta(\alpha, s) \vee \alpha \,\mathsf{R}\, \beta)$$

As a note for the reader unfamiliar with BAAs, the transition function for these automata takes a state and a symbol and returns a positive Boolean formula over the set of states Q. Furthermore, a *run* of a BAA over an infinite string $\pi = s_1 s_2 \ldots$ is characterized by a tree with labeled nodes, in which (informally): (1) the root node is labeled with the initial state, (2) level i corresponds to the processing of symbol s_i, and (3) the children of a node labeled by q at level i are the states appearing in a minimal model of $\delta(q, s_i)$. As such, multiple runs for a certain infinite string are produced when selecting different models of $\delta(q, s_i)$. A special case is when $\delta(q, s_i)$ reduces to $\top$ or $\bot$, where there is one child labeled by $\top$ or $\bot$, respectively. A run of a BAA is *accepting* iff all of its finite branches end on $\top$ and in each of its infinite branches there is an accepting state that repeats infinitely often. Figure 1 shows a run of the BAA for $\Box\Diamond p \wedge \Box\Diamond\neg p$—a formula whose semantics forces an infinite alternation, which is not necessarily immediate, between states that satisfy p and states that do not satisfy p.

In our BAA translation for LTL-FOND we follow a similar approach to that developed in the TB15 translation: given an input problem $\mathcal{P}$, we generate an equivalent problem $\mathcal{P}'$ in which we represent the configuration of the BAA with fluents (one fluent q per each state q of the BAA). $\mathcal{P}'$ contains the actions in $\mathcal{P}$ plus additional *synchronization actions* whose objective is to update the configuration of the BAA. In $\mathcal{P}'$, there are special fluents to alternate between so-called *world mode*, in which only one action of $\mathcal{P}$ is allowed, and *synchronization mode*, in which the configuration of the BAA is updated.

Before providing details of the translation we overview the main differences between our translation and that of TB15. TB15 recognizes an accepting run (i.e., a satisfied goal) by observing that all automaton states at the last level of the (finite) run are accepting states. In the infinite case, such a check does not work. As can be seen in the example of Figure 1, there is no single level of the (infinite) run that only contains final BAA states. Thus, when building a plan with our translation, the planner is given the ability to "decide" at any moment that an accepting run can be found and then the objective is to "prove" this is the case by showing the existence of a loop or *lasso* in the plan in which any non-accepting state may turn into an accepting state. To keep track of those non-accepting states that we require to eventually "turn into" accepting states we use special fluents that we call *tokens*.

For an LTL-FOND problem $\mathcal{P} = \langle \mathcal{F}, \mathcal{I}, \varphi, \mathcal{A} \rangle$, where φ is an NNF LTL formula with BAA $\mathcal{A}_\varphi = (Q, \Sigma, \delta, q_\varphi, Q_{Fin})$, the translated FOND problem is $\mathcal{P}' = \langle \mathcal{F}', \mathcal{I}', \mathcal{G}', \mathcal{A}' \rangle$, where each component is described below.

Fluents $\mathcal{P}'$ has the same fluents as $\mathcal{P}$ plus fluents for the representation of the states of the automaton $F_Q = \{q_\psi \mid \psi \in Q\}$, and flags $\mathbf{copy}$, $\mathbf{sync}$, $\mathbf{world}$ for controlling the different modes. Finally, it includes the set $F_Q^S = \{q_\psi^S \mid \psi \in Q\}$ which are *copies* of the automata fluents, and *tokens* $F_Q^T = \{q_\psi^T \mid \psi \in Q\}$. We describe both sets below. Formally, $F' = F \cup F_Q \cup F_Q^S \cup F_Q^T \cup \{\mathbf{copy}, \mathbf{sync}, \mathbf{world}, \mathbf{goal}\}$.

The set of actions $\mathcal{A}'$ is the union of the sets $\mathcal{A}_w$ and $\mathcal{A}_s$ plus the *continue* action.

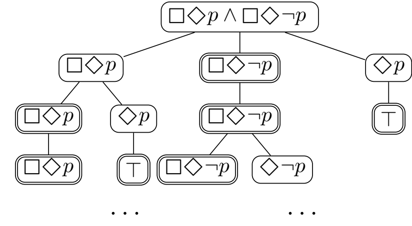


Figure 1: An accepting run of a BAA for $\Box\Diamond p \wedge \Box\Diamond\neg p$ over an infinite sequence of states in which the truth value of p alternates. Double-line ovals are accepting states/conditions.

Sync Action	Effect
$tr(q_\ell^S)$	$\{\neg q_\ell^S, q_\ell^T \rightarrow \neg q_\ell^T\}$
$tr(q_{\alpha\wedge\beta}^S)$	$\{q_\alpha^S, q_\beta^S, \neg q_{\alpha\wedge\beta}^S, q_{\alpha\wedge\beta}^T \rightarrow \{q_\alpha^T, q_\beta^T, \neg q_{\alpha\wedge\beta}^T\}\}$
$tr_1(q_{\alpha\vee\beta}^S)$	$\{q_\alpha^S, \neg q_{\alpha\vee\beta}^S, q_{\alpha\vee\beta}^T \rightarrow \{q_\alpha^T, \neg q_{\alpha\vee\beta}^T\}\}$
$tr_2(q_{\alpha\vee\beta}^S)$	$\{q_\beta^S, \neg q_{\alpha\vee\beta}^S, q_{\alpha\vee\beta}^T \rightarrow \{q_\beta^T, \neg q_{\alpha\vee\beta}^T\}\}$
$tr(q_{\bigcirc\alpha}^S)$	$\{q_\alpha, \neg q_{\bigcirc\alpha}^S, q_{\bigcirc\alpha}^T \rightarrow \{q_\alpha^T, \neg q_{\bigcirc\alpha}^T\}\}$
$tr_1(q_{\alpha\,\mathsf{U}\,\beta}^S)$	$\{q_\beta^S, \neg q_{\alpha\,\mathsf{U}\,\beta}^S, q_{\alpha\,\mathsf{U}\,\beta}^T \rightarrow \{q_\beta^T, \neg q_{\alpha\,\mathsf{U}\,\beta}^T\}\}$
$tr_2(q_{\alpha\,\mathsf{U}\,\beta}^S)$	$\{q_\alpha^S, q_{\alpha\,\mathsf{U}\,\beta}, \neg q_{\alpha\,\mathsf{U}\,\beta}^S, q_{\alpha\,\mathsf{U}\,\beta}^T \rightarrow q_\alpha^T\}$
$tr_1(q_{\alpha\,\mathsf{R}\,\beta}^S)$	$\{q_\beta^S, q_\alpha^S, \neg q_{\alpha\,\mathsf{R}\,\beta}^S, q_{\alpha\,\mathsf{R}\,\beta}^T \rightarrow \neg q_{\alpha\,\mathsf{R}\,\beta}^T\}$
$tr_2(q_{\alpha\,\mathsf{R}\,\beta}^S)$	$\{q_\beta^S, q_{\alpha\,\mathsf{R}\,\beta}, \neg q_{\alpha\,\mathsf{R}\,\beta}^S, q_{\alpha\,\mathsf{R}\,\beta}^T \rightarrow \neg q_{\alpha\,\mathsf{R}\,\beta}^T\}$
$tr_1(q_{\Diamond\alpha}^S)$	$\{q_\alpha^S, \neg q_{\Diamond\alpha}^S, q_{\Diamond\alpha}^T \rightarrow \{q_\alpha^T, \neg q_{\Diamond\alpha}^T\}\}$
$tr_2(q_{\Diamond\alpha}^S)$	$\{q_{\Diamond\alpha}, \neg q_{\Diamond\alpha}^S\}$
$tr(q_{\Box\alpha}^S)$	$\{q_\alpha^S, q_{\Box\alpha}, \neg q_{\Box\alpha}^S, q_{\Box\alpha}^T \rightarrow \neg q_{\Box\alpha}^T\}$

Table 2: Synchronization actions. The precondition of $tr(q_\psi^S)$ is $\{\mathbf{sync}, q_\psi^S\}$, plus ℓ when $\psi = \ell$ is a literal.

World Mode $\mathcal{A}_w$ contains the actions in $\mathcal{A}$ with preconditions modified to allow execution only in *world* mode. Effects are modified to allow the execution of the *copy* action, which initiates the synchronization phase, described below. Formally, $\mathcal{A}_w = \{a' \mid a \in \mathcal{A}\}$, and for all a' in $\mathcal{A}_w$:

$$Pre_{a'} = Pre_a \cup \{\mathbf{world}\},$$
$$\mathit{Eff}_{a'} = \mathit{Eff}_a \cup \{\mathbf{copy}, \neg\mathbf{world}\}.$$

Synchronization Mode This mode has three phases. In the first phase, the *copy* action is executed, adding a copy q^S for each fluent q that is currently true, deleting q. Intuitively, q^S defines the state of the automaton prior to synchronization. The precondition of *copy* is $\{\mathbf{copy}\}$, while its effect is:

$$\mathit{Eff}_{copy} = \{q \to \{q^S, \neg q\} \mid q \in F_Q\} \cup \{\mathbf{sync}, \neg\mathbf{copy}\}$$

As soon as the **sync** fluent becomes true, the second phase of synchronization begins. Here the only executable actions are those that update the state of the automaton, which are defined in Table 2. These actions update the state of the automaton following the definition of the transition function, δ. In addition, each synchronization action for a formula ψ that has an associated token q^T_ψ, *propagates* such a token to its subformulae, unless ψ corresponds to either an accepting state (i.e., ψ is of the form $\alpha \mathsf{R} \beta$) or to a literal ℓ whose truth can be verified with respect to the current state via action $tr(q^S_\ell)$.

When no more synchronization actions are possible, we enter the third phase of synchronization. Here only two actions are executable: $world$ and $continue$. The objective of $world$ action is to reestablish world mode. Its precondition is $\{\mathbf{sync}\} \cup \overline{F^S_Q}$, and its effect is $\{\mathbf{world}, \neg\mathbf{sync}\}$.

The $continue$ action also reestablishes world mode, but in addition "decides" that an accepting BAA can be reached in the future. This is reflected by the non-deterministic effect that makes the fluent **goal** true. As such, it "tokenizes" all states that are not final states in F_Q, by adding q^T for each BAA state q that is non-final and currently true. Formally,

$$Pre_{continue} = \{\mathbf{sync}\} \cup \{\neg q^T_\varphi \mid \varphi \notin Q_{Fin}\}$$
$$\mathit{Eff}_{continue} = \{\{\mathbf{goal}\},$$
$$\{q_\varphi \to q^T_\varphi \mid \varphi \notin Q_{Fin}\} \cup \{\mathbf{world}, \neg\mathbf{sync}\}\}$$

The set $\mathcal{A}_s$ is defined as the one containing actions *copy*, $world$, and all actions defined in Table 2.

Initial and Goal States The resulting problem $\mathcal{P}'$ has initial state $I' = I \cup \{q_\varphi, \mathbf{copy}\}$, and goal $\mathcal{G}' = \{\mathbf{goal}\}$.

In summary, our BAA-based approach builds on TB15 while integrating ideas from PLG13. Like PLG13 our approach uses a $continue$ action to find plans with lassos, but unlike PLG13, our translation does not directly use the accepting configuration of the automaton. Rather, the planner "guesses" that such a configuration can be reached. The token fluents F^T_Q, which did not exist in TB15, are created for each non-accepting state and can only be eliminated when a non-accepting BAA state becomes accepting.

Now we show how, given a strong cyclic policy for $\mathcal{P}'$, we can generate an FSC for $\mathcal{P}$. Observe that every state ξ, which is a set of fluents in F', can be written as the disjoint union of sets $s_w = \xi \cap F$ and $s_q = \xi \cap (F' \setminus F)$. Abusing notation, we use $s_w \in 2^F$ to represent a state in $\mathcal{P}$. For a planning state $\xi = s_w \cup s_q$ green in which $p(\xi)$ is defined, we define $\Omega(\xi)$ to be the action in $\mathcal{A}$ whose translation is $p(\xi)$. Recall now that executions of a strong-cyclic policy p for $\mathcal{P}'$ in state ξ generate plans of the form $a_1\alpha_1a_2\alpha_2\ldots$ where each a_i is a world action in $\mathcal{A}_w$ and α_i are sequences of actions in $\mathcal{A}' \setminus \mathcal{A}_w$. Thus $\Omega(\xi)$ can be generated by taking out the fluents **world** and **copy** from the precondition and effects of $p(\xi)$. If state s'_w is a result of applying $\Omega(\xi)$ in s_w, we define $\rho(\xi, s'_w)$ to be the state ξ' that results from the composition of consecutive non-world actions α_1 mandated by an execution of p in $s'_w \cup s_q$. Despite non-determinism in the executions, the state $\xi' = \rho(\xi, s'_w)$ is well-defined.

The BAA translation for LTL-FOND is sound and complete. Throughout the paper, the soundness property guarantees that FSCs obtained from solutions to the compiled problem $\mathcal{P}'$ are solutions to the LTL-FOND problem $\mathcal{P}$, whereas the completeness property guarantees that a solution to $\mathcal{P}'$ exists if one exists for $\mathcal{P}$.

Theorem 1. *The BAA translation for Infinite* LTL-*FOND planning is sound, complete, and linear in the size of the goal formula.*

A complete proof is not included but we present some of the intuitions our proof builds on. Consider a policy p' for $\mathcal{P}'$. p' yields three types of executions: (1) finite executions that end in a state where **goal** is true, (2) infinite executions in which the *continue* action is executed infinitely often and (3) infinite, unfair executions. We do not need to consider (3) because of Definition 2. Because the precondition of *continue* does not admit token fluents, if *continue* executes infinitely often we can guarantee that any state that was not a BAA accepting state turns into an accepting state. This in turn means that every branch of the run contains an infinite repetition of final states. The plan for $\mathcal{P}$, p, is obtained by removing all synchronization actions from p', and the FSC that is solution to $\mathcal{P}$ is obtained as described above. In the other direction, a plan p' for $\mathcal{P}'$ can be built from a plan p for $\mathcal{P}$ by adding synchronization actions. Theorem 1 follows from the argument given above and reuses most of the argument that TB15 uses to show their translation is correct.

3.1.2 An NBA-based Compilation This compilation relies on the construction of a non-determinisitic Büchi automaton (NBA) for the goal formula, and builds on translation techniques for *finite* LTL planning with *deterministic* actions developed by Baier and McIlraith (2006) (henceforth, BM06). Given a deterministic planning problem $\mathcal{P}$ with LTL goal φ, the BM06 translation runs in two phases: first, φ is transformed into a non-deterministic finite-state automata (NFA), $\mathcal{A}_\varphi$, such that it accepts a finite sequence of states σ if and only if $\sigma \models \varphi$. In the second phase, it builds an output problem $\mathcal{P}'$ that contains the same fluents as in $\mathcal{P}$ plus additional fluents of the form F_q, for each state q of $\mathcal{A}_\varphi$. Problem $\mathcal{P}'$ contains the same actions as in $\mathcal{P}$ but each action may contain additional effects which model the dynamics of the F_q fluents. The goal of $\mathcal{P}'$ is defined as the

disjunction of all fluents of the form F_f, where f is an accepting state of $\mathcal{A}_\varphi$. The initial state of $\mathcal{P}'$ contains F_q iff q is a state that $\mathcal{A}_\varphi$ would reach after processing the initial state of $\mathcal{P}$. The most important property of BM06 is the following: let $\sigma = s_0 s_1 \dots s_{n+1}$ be a state trace induced by some sequence of actions $a_0 a_1 \dots a_n$ in $\mathcal{P}'$, then F_q is satisfied by s_{n+1} iff there exists a run of $\mathcal{A}_\varphi$ over σ that ends in q. This means that a single sequence of planning states encodes *all* runs of the NFA $\mathcal{A}_\varphi$. The important consequence of this property is that the angelic semantics of $\mathcal{A}_\varphi$ is immediately reflected in the planning states and does not need to be handled by the planner (unlike TB15).

For LTL-FOND problem $\mathcal{P} = \langle \mathcal{F}, \mathcal{I}, \varphi, \mathcal{A} \rangle$, our NBA-based compilation constructs a FOND problem $\mathcal{P}' = \langle \mathcal{F}', \mathcal{I}', \mathcal{G}', \mathcal{A}' \rangle$ via the following three phases: (i) construct an NBA, $\mathcal{A}_\varphi$ for the NNF LTL goal formula φ, (ii) apply the *modified* BM06 translation to the determinization of $\mathcal{P}$ (see Section 2.1) , and (iii) construct the final FOND problem $\mathcal{P}'$ by undoing the determinization, i.e., reconstruct the original non-deterministic actions from their determinized counterparts. More precisely, the translation of a non-deterministic action a in $\mathcal{P}$ is a non-deterministic action a' in $\mathcal{P}'$ that is constructed by first determinizing a into a set of actions, a_i that correspond to each of the non-deterministic outcomes of a, applying the BM06-based translation to each a_i to produce a_i', and then reassembling the a_i's back into a non-deterministic action, a'. In so doing, $\mathit{Eff}_{a'}$ is the set of outcomes in each of the deterministic actions a_i', and $Pre_{a'}$ is similarly the precondition of any of these a_i'.

The modification of the BM06 translation used in the second phase leverages ideas present in PLG13 and our BAA-based compilations to capture infinite runs via induced non-determinism. In particular, it includes a $continue$ action whose precondition is the accepting configuration of the NBA (a disjunction of the fluents representing accepting states). Unlike our BAA-based compilation, the tokenization is not required because accepting runs are those that achieve accepting states infinitely often, no matter which ones. As before, one non-deterministic effect of $continue$ is to achieve **goal**, while the other is to force the planner to perform at least one action. This is ensured by adding an extra precondition to $continue$, **can_continue**, which is true in the initial state, it is made true by every action but $continue$, and is deleted by $continue$.

In order to construct a solution Π to $\mathcal{P}$ from a strong-cyclic solution p to $\mathcal{P}' = \langle \mathcal{F}', \mathcal{I}', \mathcal{G}', \mathcal{A}' \rangle$, it is useful to represent states ξ in $\mathcal{P}'$ as the disjoint union of $s = \xi \cap F$ and $q = \xi \cap (F' \setminus F)$. Intuitively, s represents the planning state in $\mathcal{P}$, and q represents the automaton state. The controller $\Pi = \langle C, c_0, \Gamma, \Lambda, \rho, \Omega \rangle$ is defined as follows. $c_0 = I'$ is the initial controller state; $\Gamma = 2^{\mathcal{F}}$; $\Lambda = \mathcal{A}$; $\rho(\xi, s') = s' \cup q'$, where q' is the automaton state that results from applying action $p(\xi)$ in ξ; $\Omega(\xi) = p(\xi)$; and $C \subseteq 2^{\mathcal{F}'}$ is the domain of p. Actions in $\mathcal{P}'$ are non-deterministic and have conditional effects, but the automaton state q' that results from applying action $p(\xi)$ in state $\xi = s \cup q$ is deterministic, and thus ρ is well-defined.

Theorem 2. *The NBA translation for* Infinite LTL-*FOND planning is sound, complete, and worst-case exponential in the size of the* LTL *formula.*

Theorem 2 follows from soundness, completeness, and the complexity of the BM06 translation, this time using a NBA automaton, and an argument similar to that of Theorem 1. This time, if *continue* executes infinitely often we can guarantee accepting NBA states are reached infinitely often.

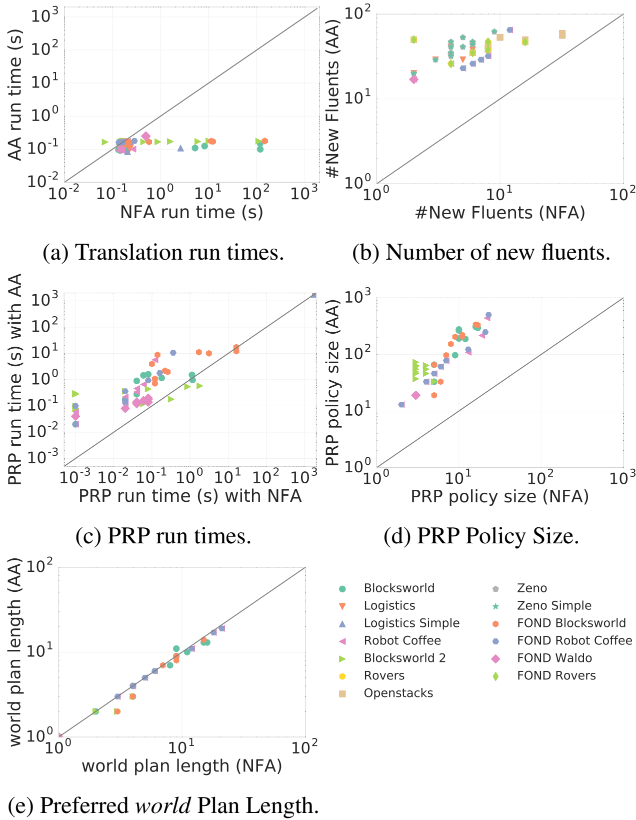
3.2 From Finite LTL-FOND to FOND

Our approach to finite LTL-FOND extends the BM06 and TB15 translations, originally intended for *finite* LTL planning with *deterministic* actions, to the non-deterministic action setting. Both the original BM06 and TB15 translations share two general steps. In step one, the LTL goal formula is translated to an automaton/automata – in the case of BM06 an NFA, in the case of TB15, an AA. In step two, a planning problem $\mathcal{P}'$ is constructed by augmenting $\mathcal{P}$ with additional fluents and action effects to account for the integration of the automaton. In the case of BM06 these capture the state of the automaton and how domain actions cause the state of the automaton to be updated. In the case of the TB15 translation, $\mathcal{P}$ must also be augmented with synchronization actions. Finally, in both cases the original problem goals must be modified to capture the accepting states of automata.

When BM06 and TB15 are exploited for LTL-FOND, the non-deterministic nature of the actions must be taken into account. This is done in much the same as with the NBA- and BAA-based compilations described in the previous section. In particular, non-deterministic actions in the LTL-FOND problem are determinized, the BM06 (resp. TB15) translation is applied to these determinized actions, and then the non-deterministic actions reconstructed from their translated determinized counterparts (as done in the NBA-based compilation) to produce FOND problem, $\mathcal{P}'$. A FSC solution, Π, to the LTL-FOND problem $\mathcal{P}$, can be obtained from a solution to $\mathcal{P}'$. When the NFA-based translations are used, the FSC, Π, is obtained from policy p following the approach described for NBA-based translations. When the AA-based translations are used, the FSC, Π, is obtained from p following the approach described for BAA-based translations.

Theorem 3. *The NFA (resp. AA) translation for Finite* LTL-*FOND is sound, complete, and exponential (resp. linear) in the size of the* LTL *formula.*

Soundness and completeness in Theorem 3 follows from soundness and completeness of the BM06 and TB15 translations. Fair executions of Π yield finite plans for $\mathcal{P}'$, and therefore state traces (excluding intermediate synchronization states) satisfy φ. Conversely, our approach is complete as for every plan in $\mathcal{P}$, one can construct a plan in $\mathcal{P}'$. Finally, the run-time complexity and size of the translations is that of the original BM06 and TB15 translations – worst case exponential in time and space for the NFA-based approach and linear in time and space for the AA approach.



(a) Translation run times. (b) Number of new fluents.

(c) PRP run times. (d) PRP Policy Size.

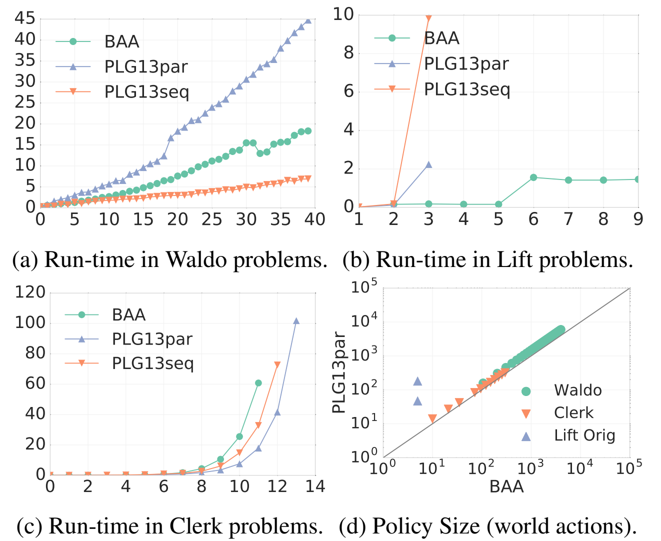
(e) Preferred *world* Plan Length.

Figure 2: Performance of our planning system using AA- and NFA-based translations in different problems with deterministic and non-deterministic actions and *finite* LTL goals.

4 Experiments

We evaluate our framework on a selection of benchmark domains with LTL goals from (Baier and McIlraith 2006; Patrizi, Lipovetzky, and Geffner 2013; Torres and Baier 2015), modified to include non-deterministic actions. Experiments were conducted on an Intel Xeon E5-2430 CPU @2.2GHz Linux server, using a 4GB memory and a 30-minute time limit.

LTL-FOND Planning over Finite Traces: We evaluated the performance of our BM06 (NFA) and TB15 (AA) translators, with respect to a collection of problems with deterministic and non-determinisitic actions and LTL goals, interpreted on finite traces. We used the state-of-the-art FOND planner, PRP (Muise, McIlraith, and Beck 2012), to solve the translated problems. NFA-based translation times increased when the LTL formula had a large number of conjunctions and nested modal operators, whereas AA-based translation times remain negligible. However, the AA translation included a number of new fluents that were, in some cases, up to one order of magnitude larger than with the NFA (Figures 2a and 2b). This seems to translate into more complex problems, as PRP run times become almost consistently greater in problems translated with AA (Figure 2c). The size of the policies obtained from the AA compilations were considerably greater than those obtained with NFA compilations (Figure 3d). This is expected, as AA translations introduce a number of synchronization actions, whereas the number of actions in NFA translations remains unchanged. To assess the quality of the plans obtained from each translation, we compared the number of *world* actions (i.e., excluding automaton-state *synchronization* actions) in the shortest plans of the policies obtained (Figure 2e). This is a crude estimator of the quality of plans, since these plans are not necessarily the ones that minimize the number of world actions, as they also contain synchronization actions. The number of world actions that we obtained in both compilations was very similar.



(a) Run-time in Waldo problems. (b) Run-time in Lift problems.

(c) Run-time in Clerk problems. (d) Policy Size (world actions).

Figure 3: Performance of our planning system using BAA-based translations in different LTL-FOND domains. We report PRP run-times (in seconds) and policy sizes, excluding synchronization actions.

Interestingly, whereas the size of the AA translations is linear in the size of the original LTL formula and NFA translations are worst-case exponential, in practice we observed the size of the NFA-based translated problems is smaller. Furthermore, PRP performs better when problems are compiled using NFAs, generating similar quality policies in lower search run-times.

We didn't experience any untoward decrease in performance in deterministic problems that were extended with non-deterministic actions, suggesting that AA- and NFA-based translations remain competitive in LTL-FOND.

LTL-FOND Planning over Infinite Traces: The relative performance observed between NBA- and BAA-based translations for LTL-FOND planning, interpreted over infinite traces, is reflective of the finite case. NBA translation run times are greater, but result in lower planner run times and smaller policy sizes. For reference, we compared BAA translations with the so-called *sequential* and *parallel* translations developed by Patrizi, Lipovetzky, and Geffner (2013), subsequently referrd to as PLG13seq and PLG13par, respectively. The former alternates between world and sync actions (that update the automaton state),

whereas the latter parallelizes this process in a single action. The current implementation of PLG13 translations forced us to perform such comparisons only in the three domains that appear in (Patrizi, Lipovetzky, and Geffner 2013). Namely, the *Waldo*, *Lift*, and *Clerk* domains. All problems have LTL goals that can be compiled into deterministic Büchi automata. Unfortunately, we could not include a fair comparison with NBA translations in the *Lift* and *Clerk* domains, due to a specific encoding that forced transitions to synchronization phases (existing in PLG13 and BAA translations, but not in NBA). In the *Waldo* problems, however, NBA translations generated smaller solutions (by a half) with roughly half the run time required by BAA. On the other hand, NBA translation times timed out after the twelfth instance (possibly due to an unoptimized implementation of the translator).

The *Waldo* problems require construction of a controller for a robot that moves around n rooms and finds Waldo infinitely often. Waldo may or may not appear in the n-th and $n/2$-th rooms when these are visited. The dynamics of the problem preclude visiting a room twice before visiting the remaining ones, in which case the predicate $search_again$ becomes true. The LTL goal of the problem is $\Box\Diamond search_again \vee Waldo$. The *Lift* problems requires construction of a controller for an n-floor building that serves all requests. The dynamics of the problem require alternation between $move$ and $push_f_i$ actions, $i = 1, \ldots, n$. Fluents at_i and req_i model, respectively, whether the lift is at the i-th floor, and whether a request from the i-th floor has been issued and not served. The lift can only move up if some request is issued. The $push_f_i$ actions non-deterministically request the lift to service the i-th floor. Initially, the lift is at floor 1, and no request is issued. The LTL goal of the problem is $\varphi = \bigwedge_{i=1}^{n} \Box\Diamond(req_i \rightarrow at_i)$. Finally, the *Clerk* problems require construction of a controller that serves all clients in a store. Clients can order one of n packages p_i. If the package is not available, the clerk has to buy it from a supplier, pick it up, and store it in its correct location. In order to serve the client, the clerk has to find the package, pick it up, and sell it. The LTL goal of the problem is $\Box(active_request \rightarrow \Diamond(item_served \vee item_stored))$.

The results of experiments are summarized in Figure 3. In *Waldo* problems, the planner run times using BAA-based translations are situated between the run times with PLG13seq and PLG13par. In *Lift* problems, the BAA translations demonstrate significantly greater scalability. The *Lift* problems contain a (increasing) large number of conjunctive LTL goals. We conjecture that the poor scalability with PLG13seq (runs out of time) and PLG13par (runs out of memory) translations is due to the bad handling of conjunctive goals, that results in a exponentially large number of different state transitions. On the other hand, the PRP handles conjunctive goals much better in the BAA translations thanks to the AA progression of the LTL formula. In the *Clerk* problems, PRP scales slightly worse with the BAA translation than with the PLG13seq and PLG13par translations, which can solve 1 and 2 more problems respectively. The run times with all translations seem to show the same exponential trend, and differ in a small offset that corresponds to the increase in problem complexity.

Figure 3d compares the size of the policies found by PRP to problems compiled with BAA and PLG13par translations. PLG13seq translations resulted in slightly larger policies, due to separate world and sync action phases. We account only for world actions, excluding synchronization actions from the count. Policy sizes with BAA-based translations are similar, but consistently smaller than those from PLG13par translations, except in the *Lift* problems where the former results in considerably smaller policies. Finally, we evaluated the validity of our system with LTL goals that could not be handled by PLG13. In particular, we solved Waldo problems with goals of the form $\Diamond\Box\alpha$.

Overall, our system proves very competitive with (as good as or better than) the previous state-of-the-art LTL-FOND planning methods, while supporting a much broader spectrum (the full spectrum) of LTL formulae.

5 Summary and Discussion

We have proposed four compilation-based approaches to fully observable non-deterministic planning with LTL goals that are interpreted over either finite or infinite traces. These compilations support the full expressivity of LTL, in contrast to much existing work. In doing so, we address a number of open problems in planning with LTL with non-deterministic actions, as noted in Table 1. Our LTL planning techniques are directly applicable to a number of real-world planning problems that are not captured by existing systems. Furthermore they are useful in a diversity of applications beyond standard planning, including but not limited to genomic rearrangement (Uras and Erdem 2010), program test generation (Razavi, Farzan, and McIlraith 2014), story generation (Haslum 2012), automated diagnosis (Grastien et al. 2007; Sohrabi, Baier, and McIlraith 2010), business process management (De Giacomo et al. 2014) and verification (Albarghouthi, Baier, and McIlraith 2009; Patrizi et al. 2011).

We evaluated the effectiveness of our FOND compilations using the state-of-the-art FOND planner, PRP. An interesting observation is that our worst-case exponential NFA-based translations run faster and return smaller policies than the AA-based linear translations. This seems to be due to the larger number of fluents (and actions) required in the AA-based translations. Compared to the existing approach of (Patrizi, Lipovetzky, and Geffner 2013), experiments indicate that our approaches scale up better.

Finally, we observe that LTL-FOND is related to the problem of LTL synthesis (Pnueli and Rosner 1989). Informally, it is the problem of computing a policy that satisfies an LTL formula, assuming that an adversary (which we can associate to the non-deterministic environment) may change some fluents after the execution of each action. Recently De Giacomo and Vardi (2015) showed how to map a finite LTL-FOND problem into a synthesis problem. Sardiña and D'Ippolito (2015) go further, showing how FOND plans can be synthesized using LTL synthesis algorithms. An open question is whether any existing planning technology can be used for LTL synthesis as well. LTL synthesis is not an instance of strong cyclic FOND planning since synthesis adversaries are not fair.

Acknowledgements: The authors gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC) and from Fondecyt grant number 1150328.

References

Albarghouthi, A.; Baier, J. A.; and McIlraith, S. A. 2009. On the use of planning technology for verification. In *Proceedings of the Validation and Verification of Planning and Scheduling Systems Workshop (VVPS)*.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *AI Magazine* 16:123–191.

Baier, J. A., and McIlraith, S. A. 2006. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI)*, 788–795.

Cresswell, S., and Coddington, A. M. 2004. Compilation of LTL goal formulas into PDDL. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI)*, 985–986.

De Giacomo, G., and Vardi, M. Y. 2013. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, 854–860.

De Giacomo, G., and Vardi, M. Y. 2015. Synthesis for LTL and LDL on finite traces. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 1558–1564.

De Giacomo, G.; Masellis, R. D.; Grasso, M.; Maggi, F. M.; and Montali, M. 2014. Monitoring business metaconstraints based on LTL and LDL for finite traces. In *Proceedings of the 12th International Conference on Business Process Management (BPM)*, volume 8659 of *Lecture notes in Computer Science*, 1–17. Springer.

De Giacomo, G.; Masellis, R. D.; and Montali, M. 2014. Reasoning on LTL on finite traces: Insensitivity to infiniteness. In *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI)*, 1027–1033.

Doherty, P., and Kvarnström, J. 2001. TALplanner: A temporal logic-based planner. *AI Magazine* 22(3):95–102.

Edelkamp, S. 2006. Optimal symbolic PDDL3 planning with MIPS-BDD. In *5th International Planning Competition Booklet (IPC-2006)*, 31–33.

Geffner, H., and Bonet, B. 2013. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences for PDDL3. Technical Report 2005-08-07, Department of Electronics for Automation, University of Brescia, Brescia, Italy.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated planning: theory & practice*. Elsevier.

Grastien, A.; Anbulagan; Rintanen, J.; and Kelareva, E. 2007. Diagnosis of discrete-event systems using satisfiability algorithms. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI)*, 305–310.

Haslum, P. 2012. Narrative planning: Compilations to classical planning. *Journal of Artificial Intelligence Research* 44:383–395.

Kabanza, F.; Barbeau, M.; and St.-Denis, R. 1997. Planning control rules for reactive agents. *Artificial Intelligence* 95(1):67–11.

Muise, C.; McIlraith, S. A.; and Beck, J. C. 2012. Improved Non-deterministic Planning by Exploiting State Relevance. In *Proceedings of the 22th International Conference on Automated Planning and Scheduling (ICAPS)*, 172–180.

Patrizi, F.; Lipovetzky, N.; De Giacomo, G.; and Geffner, H. 2011. Computing infinite plans for LTL goals using a classical planner. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, 2003–2008.

Patrizi, F.; Lipovetzky, N.; and Geffner, H. 2013. Fair LTL synthesis for non-deterministic systems using strong cyclic planners. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, 2343–2349.

Pistore, M., and Traverso, P. 2001. Planning as model checking for extended goals in non-deterministic domains. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, 479–484.

Pnueli, A., and Rosner, R. 1989. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 179–190.

Pnueli, A. 1977. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science (FOCS)*, 46–57.

Razavi, N.; Farzan, A.; and McIlraith, S. A. 2014. Generating effective tests for concurrent programs via AI automated planning techniques. *International Journal on Software Tools for Technology Transfer* 16(1):49–65.

Rintanen, J. 2000. Incorporation of temporal logic control into plan operators. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI)*, 526–530.

Sardiña, S., and D'Ippolito, N. 2015. Towards fully observable non-deterministic planning as assumption-based automatic synthesis. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 3200–3206.

Sohrabi, S.; Baier, J. A.; and McIlraith, S. A. 2010. Diagnosis as planning revisited. In *Proceedings of the 12th International Conference on the Principles of Knowledge Representation and Reasoning (KR)*, 26–36.

Torres, J., and Baier, J. A. 2015. Polynomial-time reformulations of LTL temporally extended goals into final-state goals. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI)*, 1696–1703.

Uras, T., and Erdem, E. 2010. Genome rearrangement: A planning approach. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI)*.

Vardi, M. Y., and Wolper, P. 1994. Reasoning about infinite computations. *Information and Computation* 115(1):1–37.

Vardi, M. Y. 1995. An automata-theoretic approach to linear temporal logic. In *Banff Higher Order Workshop*, volume 1043 of *Lecture notes in Computer Science*, 238–266. Springer.

Abstraction Heuristics for Symbolic Bidirectional Search

Álvaro Torralba
Saarland University
Saarbrücken, Germany
torralba@cs.uni-saarland.de

Carlos Linares López and **Daniel Borrajo**
Universidad Carlos III de Madrid
Madrid, Spain
{carlos.linares,daniel.borrajo}@uc3m.es

Abstract

Symbolic bidirectional uniform-cost search is a prominent technique for cost-optimal planning. Thus, the question whether it can be further improved by making use of heuristic functions raises naturally. However, the use of heuristics in bidirectional search does not always improve its performance. We propose a novel way to use abstraction heuristics in symbolic bidirectional search in which the search only resorts to heuristics when it becomes unfeasible. We adapt the definition of partial and perimeter abstractions to bidirectional search, where A* is used to traverse the abstract state spaces and/or generate the perimeter. The results show that abstraction heuristics can further improve symbolic bidirectional search in some domains. In fact, the resulting planner, SymBA*, was the winner of the optimal-track of the last IPC.

Introduction

Most cost-optimal planners are based on A* guided with an admissible heuristic. Bidirectional search has not been explored so extensively, due to the inherent difficulties of regression in planning and the computational cost of detecting collisions between both frontiers (Alcázar, Fernández, and Borrajo 2014). Symbolic search (McMillan 1993) reasons over sets of states, substantially reducing the cost of detecting the collision of frontiers. Besides, recent advances have alleviated the problem of spurious states in symbolic regression (Torralba and Alcázar 2013). Thus, symbolic bidirectional uniform-cost search (SB) is among the best algorithms for cost-optimal planning, outperforming not only A*-based planners but also BDDA*, the symbolic search variant of A*.

These observations lead to the question of whether heuristics can further improve SB. Bidirectional heuristic search (BHS) has a long history (Pohl 1969; de Champeaux 1983), but it has never convincingly outperformed A* across a significant number of domains. There have been various attempts to explain the main reasons behind the limitations of front-to-end BHS, from the search frontiers passing each other without intersecting (Nilsson 1982) to the hardness of proving optimality (Kaindl and Kainz 1997). A recent study conjectures that the quality of heuristics is a major factor for explaining the disappointing empirical results of BHS (Barker and Korf 2015) and motivating new approaches (Holte et al. 2016).

Perimeter search is a variant of BHS that creates a perimeter around the goal, and uses heuristics that estimate the distance to the perimeter instead of to the goal (Dillenburg and Nelson 1994). Abstraction heuristics are a good fit because they precompute the heuristic, avoiding a large overhead during the search (Eyerich and Helmert 2013). Moreover, symbolic perimeter abstraction heuristics are state-of-the-art for cost-optimal planning (Torralba, Linares López, and Borrajo 2013).

We present a new planner, SymBA*, that combines symbolic bidirectional search with perimeter abstraction heuristics, exploiting their synergy to benefit from the advantages of BHS and overcome its limitations. SymBA* performs bidirectional searches over different state spaces. It starts in the original search space and, when the search becomes too hard, it derives a perimeter abstraction heuristic. The planner decides at any point whether to advance the search in the original state space, enlarging the perimeter, or in an abstract space to improve the heuristic. To that end, we introduce a new type of abstraction heuristics that uses bidirectional search combined with perimeter and partial abstractions. This is the first time bidirectional search is used to explore abstract state spaces to the best of the authors' knowledge. Even though the theory behind partial and perimeter abstractions has been well studied, they have to be adapted for their combination with bidirectional search. In particular, we study how partial abstractions can be used when A* search is used to traverse the abstract state space and how the initialization of perimeter abstractions can be improved in the bidirectional setting.

Preliminaries

A *planning task* is a 4-tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$. $\mathcal{V}$ is a finite set of *variables* v, each $v \in \mathcal{V}$ being associated with a finite domain D_v. A *partial state* over $\mathcal{V}$ is a function s on a subset $V(s)$ of $\mathcal{V}$, so that $s(v) \in D_v$ for all $v \in V(s)$; s is a *state* if $V(s) = \mathcal{V}$. The *initial state* $\mathcal{I}$ is a state. The *goal* $\mathcal{G}$ is a partial state. $\mathcal{A}$ is a finite set of *actions*, each $a \in \mathcal{A}$ being a pair (pre_a, eff_a) of partial states, called its *precondition* and *effect*. Each $a \in \mathcal{A}$ has a non-negative *cost*, $c(a) \in \mathbb{R}^+_0$.

The *state space* of a planning task Π is a labeled transition system $\Theta^\Pi = (S, L, T, s_0, S_G)$ where: S is the set of all states; s_0 is the initial state $\mathcal{I}$ of Π; $s \in S_G$ iff $\mathcal{G} \subseteq s$; the labels L correspond to the actions A, and $s \xrightarrow{a} t$ is a

transition in T if s complies with pre_a, and $t(v) = \mathit{eff}_a(v)$ for $v \in V(\mathit{eff}_a)$ while $t(v) = s(v)$ for $v \in V \setminus V(\mathit{eff}_a)$. A *plan* for s is a path from s to any $s_G \in S_G$. The cost of a plan is defined as $c(\pi) = \sum_{a_i \in \pi} c(a_i)$. The cost of a cheapest plan for s is denoted $h^*(s)$ and the cost of the cheapest path from s_0 to s is denoted $g^*(s)$. A plan for s_0 is a plan for Π, and is *optimal* iff its cost equals $h^*(s_0)$.

Most cost-optimal planners use heuristic search with A*. A *heuristic* is a function $h : \mathcal{S} \to \mathbb{R}_0^+ \cup \{\infty\}$ which estimates the remaining cost to reach the goal. A heuristic is *perfect* if it coincides with h^*, and it is *admissible* if it never overestimates the optimal cost, that is, $\forall s : h(s) \leq h^*(s)$. A heuristic is *consistent* if for every transition $s \xrightarrow{a} t$, $h(s) \leq h(t) + c(a)$. A* expands the nodes with lowest $f(s) = g(s) + h(s)$ first so that no node with $f(s) > h^*(\mathcal{I})$ is ever expanded. If the heuristic is admissible A* is guaranteed to return an optimal solution. Moreover, if the heuristic is consistent, A* always closes states with their optimal g-value, $g^*(s)$, so it does not re-expand any node.

Bidirectional Search

A bidirectional search, $\mathcal{T}$ is composed of a forward, $\mathcal{T}_{fw}$, and a backward, $\mathcal{T}_{bw}$, unidirectional search. We use $\mathcal{T}_u$ to denote a unidirectional search in an unspecified direction and $\mathcal{T}_{\neg u}$ for the search in the opposite direction. Each search $\mathcal{T}_u$ consists of an open, $open(\mathcal{T}_u)$, and a closed list, $closed(\mathcal{T}_u)$. We denote by $g(\mathcal{T}_u)$ and $f(\mathcal{T}_u)$ the minimum g and f-values of any state in $open(\mathcal{T}_u)$, respectively.

Nipping is an optimization that avoids expanding any state if it has already been expanded in the opposite direction (Kwa 1989). We apply nipping at generation time in order to avoid unnecessary evaluations. We rely on states always being closed with their optimal g-value, g^*, so that $g^*(s)$ is known for all $s \in closed(\mathcal{T}_u)$ and those $s \in open(\mathcal{T}_u)$ with $g^*(s) \leq g(\mathcal{T}_u) + \min_{a \in \mathcal{A}} c(a)$. Thus, whenever a state s is generated or is going to be expanded in $\mathcal{T}_u$, if $g^*(s)$ is known for $\mathcal{T}_{\neg u}$, we discard s (without introducing it in $closed(\mathcal{T}_u)$) and store the plan through that state if it is the best plan found so far. The algorithm terminates when the best solution found so far, π, has been proven optimal, i. e., $c(\pi) \leq \max(f(\mathcal{T}_{fw}), f(\mathcal{T}_{bw}))$. Both frontiers use each other as a heuristic, assigning an admissible value of $g(\mathcal{T}_{\neg u}) + \min_{a \in \mathcal{A}} c(a)$ to all states in $open(\mathcal{T}_u)$. As all states in $open(\mathcal{T}_u)$ have the same heuristic value, this does not affect to the expansion order of uniform-cost search, but allows to terminate the algorithm whenever $c(\pi) \leq g(\mathcal{T}_{bw}) + g(\mathcal{T}_{fw}) + \min_{a \in \mathcal{A}} c(a)$.

Abstraction Heuristics

An abstraction is a mapping $\alpha : \mathcal{S} \to \mathcal{S}^\alpha$ from states to abstract states. The *abstract state space* is a tuple $\Theta^\alpha = \langle \mathcal{S}^\alpha, L, T^\alpha, \mathcal{I}^\alpha, \mathcal{S}^\alpha_\star \rangle$ where $\mathcal{S}^\alpha$ is the set of abstract states, L is the set of labels, $T^\alpha = \{(\alpha(s) \xrightarrow{a} \alpha(t)) \mid s \xrightarrow{a} t\}$, $\mathcal{I}^\alpha = \alpha(s_0)$ and $\mathcal{S}^\alpha_\star = \{s^\alpha \mid \exists s \in \mathcal{S}_\star, s^\alpha = \alpha(s)\}$. $\mathcal{T}^\alpha$ denotes a bidirectional search in Θ^α, in contrast to the search in the original state space, $\mathcal{T}^\Pi$. Abstraction heuristics use the optimal solution cost in Θ^α, h^α, as an admissible estimation. Similarly, $g^\alpha(s)$ is the optimal solution cost from $\mathcal{I}^\alpha$ to $\alpha(s)$.

There are different types of abstraction heuristics depending on the mapping definition, α. Pattern Databases (PDBs) (Culberson and Schaeffer 1998; Edelkamp 2001) are projections of Π onto a subset of variables (called *pattern*), so that two states are equivalent iff they agree on the value of variables in the pattern. Merge-and-shrink (M&S) abstractions generalize PDBs, allowing abstractions that use all variables (Helmert, Haslum, and Hoffmann 2007; Helmert et al. 2014).

The optimal solution cost from every abstract state, $h^\alpha(s^\alpha)$, is precomputed and stored in a lookup table prior to the search by performing a backward uniform-cost search from the abstract goal, $\mathcal{T}^\alpha_{bw}$. Partial abstractions do not search the entire abstract state space completely (Anderson, Holte, and Schaeffer 2007; Edelkamp and Kissmann 2008b). Thus, h^α is only known for states that were expanded during the precomputation phase or were left in open with g-value lower or equal than $g(\mathcal{T}^\alpha) + \min_{a \in \mathcal{A}} c(a)$. For every other abstract state, the heuristic returns the minimum cost with which a state could be generated $g(\mathcal{T}^\alpha) + \min_{a \in \mathcal{A}} c(a)$. Partial abstractions are admissible and consistent.

Perimeter abstractions construct a perimeter around the goal in the original state space and use it to seed the search in the abstract state space (Felner and Ofek 2007; Eyerich and Helmert 2013). The perimeter is constructed by a backward search, $\mathcal{T}^\Pi_{bw}$, which computes the perfect heuristic for all states in $closed(\mathcal{T}^\Pi_{bw})$. For every state outside the perimeter, an abstract search, $\mathcal{T}^\alpha_{bw}$ computes the minimum distance from each abstract state to the closest abstract state in the perimeter. Formally, $\mathcal{T}^\alpha_{bw}$ is initialized with: $open(\mathcal{T}^\alpha_{bw})[g] = \{s^\alpha \mid \exists_{s \in S, \alpha(s) = s^\alpha}\ s \in open(\mathcal{T}^\Pi_{bw})[g]\}$ and $closed(\mathcal{T}^\alpha_{bw}) = \{s^\alpha \mid \forall_{s \in S, s^\alpha = \alpha(s)}\ s \in closed(\mathcal{T}^\Pi_{bw})\}$.

Symbolic Search

Symbolic search algorithms use succinct data-structures like Binary Decision Diagrams (Bryant 1986) to efficiently represent and manipulate sets of states. BDDs offer a compact representation of sets of states that sometimes can get an exponential advantage in memory with respect to their explicit enumeration (Edelkamp and Kissmann 2008a). Furthermore, BDD operations can be used to compute the union or intersection of two sets of states. Using these operations, it is possible to define symbolic versions of different search algorithms like uniform-cost search or A*.

BDDA* is the symbolic version of A*. As usual, it expands states in ascending order of their f-value, but expanding at the same time all the states sharing the same f and g values. A difference with typical explicit implementations of A* is that it uses the opposite tie-breaking. In BDDA* states with lower g-value are preferred in order to generate all the states with the same f and g value before expanding any of them.

Symbolic search is not limited to the original state space. Symbolic PDBs take advantage of symbolic search in order to traverse the abstract state space. This allows for the use of larger patterns, since the state space is not explicitly enumerated (Kissmann 2012). The combination of symbolic search

and perimeter abstractions is a state-of-the-art heuristic (Torralba, Linares López, and Borrajo 2013), which we extend to the bidirectional case.

SymBA*: Symbolic Bidirectional A*

SymBA* performs several symbolic bidirectional A* searches on different state spaces. First, SymBA* starts a bidirectional search in the original state space, $\mathcal{T}^{\Pi}$. At each iteration, the algorithm performs a step in a selected direction, i. e., expands the set of states with minimum f-value in the frontier. Since no abstraction heuristic has been derived yet, it behaves like symbolic bidirectional uniform-cost search. This search continues until the next layer in both directions is deemed as unfeasible, because SymBA* estimates that it will take either too much time or memory. Only then, a new bidirectional search is started in an abstract state space, $\mathcal{T}^{\alpha}$ initialized with the current frontiers of $\mathcal{T}^{\Pi}$. The abstract searches provide heuristic estimations, increasing the f-value of states in the original search frontiers. Eventually, the search in the original state space will be simplified (as the number of states with minimum f-value will be smaller)[1] and SymBA* will continue the search in the original state space.

One important feature of the algorithm is the lazy evaluation of the heuristics. The search in abstract state spaces is delayed until strictly needed to simplify the original search, allowing SymBA* to use multiple abstraction heuristics without a large overhead. We model this by considering a pool of active searches and letting the algorithm decide which search should be advanced at any step, as shown in Alg. 1. The pool of searches is initialized with a bidirectional search in the original state space. At each iteration, the algorithm filters the searches that are *valid candidates* from the pool and selects the most promising one. The algorithm depends on this *search selection strategy*, further explained in Section .

Once a search has been selected, the procedure `ExpandFrontier` expands the set of states that have a minimum g-value among those that have a minimum f-value, as usual in BDDA*. If we progress the original search, a new plan with a lower cost than the incumbent solution may be found. If an abstract search is selected, we update the heuristic value of states in searches in the opposite direction in the pool, both in the abstract and original state spaces. If several abstraction heuristics are generated, SymBA* will use their maximum value, so that the heuristic value of states can only be increased. If there are no valid search candidates (line 10), a new bidirectional search is added to the pool (which amounts to two new searches). The *abstraction strategy* relaxes the current frontiers of the original search, until the frontier size is small enough to continue the search.

One of the main characteristics of SymBA* is that the heuristics change dynamically during the search. Not only may the algorithm decide to initialize a new abstract search

[1]Having fewer states does not necessarily imply that the BDD is smaller, but in most cases there is a positive correlation.

Algorithm 1: SymBA*

Input: Planning problem: $\Pi = \langle \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$
Output: Cost-optimal plan or "no plan"
1 $\mathit{SearchPool} \leftarrow \{\mathcal{T}^{\Pi}_{fw}, \mathcal{T}^{\Pi}_{bw}\}$
2 $\pi \leftarrow$ "noplan"
3 **while** $\max(f(\mathcal{T}^{\Pi}_{fw}), f(\mathcal{T}^{\Pi}_{bw})) < \mathit{cost}(\pi)$ **do**
4   **if** $\exists \mathcal{T}^{X}_{u} \in \mathit{SearchPool}$ s.t. `Is-Candidate`($\mathcal{T}^{X}_{u}$) **then**
5     $\mathcal{T}^{X}_{u} \leftarrow$ `Select-Search`($\mathit{SearchPool}$)
6     $\pi' \leftarrow$ `Expand-frontier`($\mathcal{T}^{X}_{u}$)
7   **if** $X = \Pi \wedge \pi' \neq \emptyset \wedge \mathit{cost}(\pi') < \mathit{cost}(\pi)$ **then**
8       $\pi \leftarrow \pi'$
9   `Notify-h`($\mathcal{T}^{X}_{\neg u}, \mathcal{T}^{\Pi}_{\neg u}$)
10   **else**
11     $\alpha \leftarrow$`Select-abstraction`($\Pi, \mathcal{T}^{\Pi}_{fw}, \mathcal{T}^{\Pi}_{bw}$)
12     $\left\langle \mathcal{T}^{\alpha}_{fw}, \mathcal{T}^{\alpha}_{bw} \right\rangle \leftarrow$`Apply`($\alpha, \mathcal{T}^{\Pi}_{fw}, \mathcal{T}^{\Pi}_{bw}$)
13     $\mathit{SearchPool} \leftarrow \mathit{SearchPool} \cup \{\mathcal{T}^{\alpha}_{fw}, \mathcal{T}^{\alpha}_{bw}\}$
14 **return** π

at any point, but also every time that an abstract search performs a step, the heuristic value of states in the original search may increase. Re-evaluating the entire search frontier repeatedly may be too costly if done naïvely, becoming a bottleneck and making the entire algorithm unfeasible. We avoid this problem using the lazy implementation of BDDA* (Edelkamp, Kissmann, and Torralba 2012), which keeps the states organized by g-value and defers the heuristic evaluation. Thus, whenever the heuristic changes in the middle of the search only the set of states currently selected for expansion must be re-evaluated, without any additional computation in the open list.

The next sections describe the abstraction heuristics that we use. Summarizing, (i) bidirectional search can be used in the abstract state space allowing two searches, in opposite directions, to exchange information and avoid redundant work; (ii) partial abstractions allow SymBA* to traverse larger abstract state spaces with less effort, since exploring them completely is unnecessarily expensive; and (iii) perimeter abstractions take advantage of the searches in the original state space in order to obtain better estimates, overcoming the limitations of front-to-end BHS. Perimeter and partial abstraction heuristics are not new and the conditions for consistency and admissibility are well-known for them. However, the use of bidirectional abstract searches and, in particular, the use of A* searches for exploring the abstract state space and constructing the perimeter, requires us to reconsider partial and perimeter abstractions. Our aim is to obtain heuristic estimations as informed as possible while preserving the optimality of the algorithm. Inconsistency of heuristics is not necessarily a problem (Felner et al. 2011), but states must be closed with their optimal g^*-value to ensure that perimeter abstractions are admissible.

Bidirectional Abstractions

To perform bidirectional search in abstract state spaces, a distinction must be made between the searches used in the original and abstract state spaces. $\mathcal{T}^{\Pi}$ aims to find a plan. So, whenever the two frontiers meet, a plan is retrieved and nipping avoids the expansion of the state to eliminate redundant work between $\mathcal{T}^{\Pi}_{fw}$ and $\mathcal{T}^{\Pi}_{bw}$. On the other hand, searches on abstract state spaces are used to derive heuristic estimates for the original search. Therefore, nipping must be disabled in order for the estimations to be admissible.

Thus, the interaction between both searches is reduced to use each other as a (perfect) heuristic. But, otherwise, they do not directly interact to detect the collision of their frontiers. Hence, bidirectional searches in the abstract state space are two A* searches using each other as a heuristic. They must redundantly expand states that have already been expanded in the other direction in order to provide admissible estimations to the original search. In the worst case, if both abstract searches traverse the entire abstract state space, the search effort is doubled plus an overhead for using heuristics. Here it is where partial abstractions come in handy to avoid the exploration of the entire abstract state space.

Partial Abstractions with Heuristic Search

SymBA* uses partial abstractions to traverse large abstract state spaces that could not be entirely explored otherwise. In order to compute the heuristic value of every state, we will likely need to expand most parts of the abstract state space. For example, if one single state is a dead-end both in the original and the abstract state space, the abstract state space must be completely traversed before continuing the search. Most parts of that computation are irrelevant, since the termination criterion of A* is that, for every state s not expanded yet, $f^*(s) \geq c(\pi)$. In other words, the heuristic value of a state does not matter provided it is large enough to guarantee that its f-value is not the minimum among the current f-value of other states in the search. To simplify the notation, we assume wlog (the same arguments hold for opposite directions) that a backward abstract A* search, $\mathcal{T}^{\alpha}_{bw}$ guided with an admissible estimation of g^{α} is used to compute h^{α} in order to inform a forward search in the original state space, $\mathcal{T}^{\Pi}_{fw}$.

In partial abstractions, abstract states can be classified depending on whether $h^{\alpha}(s)$ is known or not. $h^{\alpha}(s)$ is known for those abstract states that have already been expanded and those that remain in open with a g-value lower or equal than $g(\mathcal{T}^{\alpha}_{bw}) + \min_{a\in\mathcal{A}} c(a)$. The question is which heuristic value do we assign to states for which $h^{\alpha}(s)$ is not known. The usual answer is to use the minimum g-value with which they could be generated, $g(\mathcal{T}^{\alpha}_{bw}) + \min_{a\in\mathcal{A}} c(a)$. Usually, this is a satisfactory lower bound because the abstract state space is explored with a uniform-cost search so $g(\mathcal{T}^{\alpha}_{bw})$ is constantly increasing. However, in SymBA* the abstract state space is explored with an A* search because it uses the other frontier as (perfect) heuristic. Thus, a bound only based on $g(\mathcal{T}^{\alpha}_{bw})$ is no longer useful, because it may remain constant as the search progresses. Therefore,

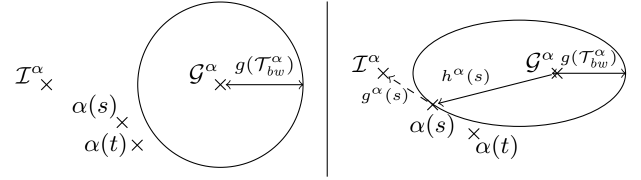


Figure 1: $\mathcal{T}^{\alpha}_{bw}$ with uniform-cost search (left) and A* (right).

states, we set a bound for the f-value of the states, based on the following inequality: $f(\mathcal{T}^{\alpha}_{bw}) \leq f^{\alpha}(s) \leq f^*(s)$. As $f(s) = g(s) + h(s)$, we can translate the bound for $f(s)$ to a heuristic value that depends on the g-value of s: $h(s) = f(\mathcal{T}^{\alpha}_{bw}) - g(s)$.

Proposition 1. *Let $\mathcal{T}^{\alpha}_{bw}$ be a search in Θ^{α} and $s \in S$ be a state s.t. $\alpha(s) \notin closed(\mathcal{T}^{\alpha}_{bw})$. Then $f(\mathcal{T}^{\alpha}_{bw}) - g(s) \leq h^*(s)$.*

Proof. By definition $g(s) \geq g^*(s)$. Also, $f(\mathcal{T}^{\alpha}_{bw}) \leq f^{\alpha}(s) \leq f^*(s) = g^*(s) + h^*(s)$. Thus, the inequality holds: $h^*(s) = f^*(s) - g^*(s) \geq f^*(s) - g(s) \geq f(\mathcal{T}^{\alpha}_{bw}) - g(s)$. □

The problem is that such heuristic may be inconsistent, as illustrated by Figure 1. Consistency requires that for every $s \xrightarrow{a} t$, $h(s) \leq h(t) + c(a)$. However, if $\alpha(s)$ has been expanded by the abstract search and $\alpha(t)$ not, the bound for $\alpha(t)$, $f(\mathcal{T}^{\alpha}_{bw}) - g(t)$ might be a lot weaker than $h^{\alpha}(s)$, generating an inconsistency. This may cause t to be expanded before s with a suboptimal g-value. In order to avoid closing any state with a suboptimal g-value, one must ensure that the lower-bound $h^{\alpha}(s)$ is only used when $f(\mathcal{T}^{\alpha}_{bw})$ is large enough to satisfy $h^{\alpha}(s) \leq f(\mathcal{T}^{\alpha}_{bw}) - g(t)$. So, the abstract search cannot be stopped at any point, it must continue until $f(\mathcal{T}^{\Pi}_{fw}) \leq f(\mathcal{T}^{\alpha}_{bw})$. We model this by computing the minimum between h^{α} and our f-based bound.

Definition 1 (Partial abstraction heuristic)**.** *Let $\mathcal{T}^{\alpha}_{bw}$ be an A* search over Θ^{α} informed with an admissible and consistent heuristic. We define the bound for unexplored states B, as $B(s, \mathcal{T}^{\alpha}_{bw}) := \max\{f(\mathcal{T}^{\alpha}_{bw}) - g(s), g(\mathcal{T}^{\alpha}_{bw}) + \min_{a\in\mathcal{A}} c(a)\}$. We define the* partial abstraction heuristic *as:*

$$h_{\mathcal{T}^{\alpha}}(s) = \begin{cases} \min(h^{\alpha}(s), B(s, \mathcal{T}^{\alpha}_{bw})) & \textit{if } h^{\alpha}(s) \textit{ is known in } \mathcal{T}^{\alpha}_{bw} \\ B(s, \mathcal{T}^{\alpha}_{bw}) & \textit{otherwise} \end{cases}$$

$h_{\mathcal{T}^{\alpha}}$ is still an inconsistent heuristic because if neither $h^{\alpha}(s)$ nor $h^{\alpha}(t)$ are known, $B(s, \mathcal{T}^{\alpha}_{bw}) = f(\mathcal{T}^{\alpha}_{bw}) - g(s)$ and $B(t, \mathcal{T}^{\alpha}_{bw}) = f(\mathcal{T}^{\alpha}_{bw}) - g(t)$ then $h(s) = f(\mathcal{T}^{\alpha}_{bw}) - g(s)$ and $h(t) = f(\mathcal{T}^{\alpha}_{bw}) - g(t)$. This is clearly inconsistent since it may be that $g(s) \ll g(t)$, since $g^*(t) \leq g^*(s) + c(a)$ but s has not been expanded yet so $g^*(t) \ll g(t)$ is possible. However, then $f(s) = f(t)$, and the one with lower g-value is selected for expansion. Theorem 2 proves this in general.

Theorem 2. *Let $\mathcal{T}^{\Pi}_{fw}$ be an A* search that breaks ties in favor of states with lower g-value*[2] *and is informed with a $h_{\mathcal{T}^{\alpha}}$. Then, for any $s \in closed(\mathcal{T}^{\Pi}_{fw})$, $g(s) = g^*(s)$.*

[2]The tie-breaking condition is not needed if $f(\mathcal{T}^{\alpha}_{bw}) > f(\mathcal{T}^{\Pi}_{fw})$.

Proof. Suppose that $g(s) > g^*(s)$. By Lemma 1 in (Hart, Nilsson, and Raphael 1968), there exists a state $r \in open(\mathcal{T}_{fw}^{\Pi})$, $r \neq s$, which is in the optimal path from $\mathcal{I}$ to s such that $g(r) = g^*(r)$. Since r lies on the optimal path from $\mathcal{I}$, $g^*(r) = g(r) \leq g^*(s) < g(s)$. On the other hand, since s was selected for expansion by $\mathcal{T}_{fw}^{\Pi}$, $f(s) < f(r)$. We prove that this leads to contradiction for the possible heuristic values of r and s.

1. $h_{\mathcal{T}^\alpha}(s) = h^\alpha(s)$.
   By the definition of $h_{\mathcal{T}^\alpha}$, $h_{\mathcal{T}^\alpha}(r) \leq h^\alpha(r)$. This is straightforward if $h^\alpha(r)$ is known. If $h^\alpha(r)$ is not known, then $g(\mathcal{T}_{bw}^\alpha) + \min_{a\in\mathcal{A}} c(a) \leq h^\alpha(r)$ because this is the minimum cost with which any state can be generated in $\mathcal{T}_{bw}^\alpha$. Moreover, $\alpha(r)$ has not been expanded by $\mathcal{T}_{bw}^\alpha$ so $f(\mathcal{T}_{bw}^\alpha) \leq f^\alpha(r) \leq g^*(r) + h^\alpha(r)$. Then, $h_{\mathcal{T}^\alpha}(r) = f(\mathcal{T}_{bw}^\alpha) - g^*(r) \leq h^\alpha(r)$.
   This leads to contradiction with consistency of h^α:

$$g^*(s) = g^*(r) + c(r,s) < g(s)$$
$$g^*(r) + c(r,s) + h^\alpha(s) < g(s) + h^\alpha(s) = f(s)$$
$$f(s) < f(r)$$
$$g^*(r) + c(r,s) + h^\alpha(s) < f(r) \leq g^*(r) + h^\alpha(r)$$

2. $h_{\mathcal{T}^\alpha}(s) = f(\mathcal{T}_{bw}^\alpha) - g(s)$. Then, $f(s) = f(\mathcal{T}_{bw}^\alpha) < f(r)$. We have two cases depending on the value of $B(r, \mathcal{T}_{bw}^\alpha)$:
   (a) $B(r, \mathcal{T}_{bw}^\alpha) = f(\mathcal{T}_{bw}^\alpha) - g(r)$. Since $h_{\mathcal{T}^\alpha}(r) \leq B(r, \mathcal{T}_{bw}^\alpha) = f(\mathcal{T}_{bw}^\alpha) - g(r)$, $f(r) \leq f(\mathcal{T}_{bw}^\alpha)$ reaching a contradiction.
   (b) $B(r, \mathcal{T}_{bw}^\alpha) = g(\mathcal{T}_{bw}^\alpha) + \min_{a\in\mathcal{A}} c(a) > f(\mathcal{T}_{bw}^\alpha) - g(r)$. $h_{\mathcal{T}^\alpha}(s) = f(\mathcal{T}_{bw}^\alpha) - g(s) \geq g(\mathcal{T}_{bw}^\alpha) + \min_{a\in\mathcal{A}} c(a) > f(\mathcal{T}_{bw}^\alpha) - g(r)$. Therefore, $g(s) < g(r) = g^*(r)$, which is not possible since r is in the optimal path to s.
3. $h_{\mathcal{T}^\alpha}(s) = g(\mathcal{T}_{bw}^\alpha) + \min_{a\in\mathcal{A}} c(a)$.
   We have two cases depending on the value of $B(r, \mathcal{T}_{bw}^\alpha)$:
   (a) $B(r, \mathcal{T}_{bw}^\alpha) = g(\mathcal{T}_{bw}^\alpha) + \min_{a\in\mathcal{A}} c(a)$.
   Since $h_{\mathcal{T}^\alpha}(r) \leq B(r, \mathcal{T}_{bw}^\alpha) = h(s)$, $f(r) < f(s)$ reaching a contradiction.
   (b) $B(r, \mathcal{T}_{bw}^\alpha) = f(\mathcal{T}_{bw}^\alpha) - g(r) > g(\mathcal{T}_{bw}^\alpha) + \min_{a\in\mathcal{A}} c(a)$. $f(\mathcal{T}_{bw}^\alpha) - g(r) > g(\mathcal{T}_{bw}^\alpha) + \min_{a\in\mathcal{A}} c(a) \geq f(\mathcal{T}_{bw}^\alpha) - g(s)$. Therefore, $g(s) < g(r) = g^*(r)$, which is not possible since r is in the optimal path to s.

Finally, we prove that tie-breaking is not needed when $f(\mathcal{T}_{bw}^\alpha) > f(\mathcal{T})$. Note that $B(s, \mathcal{T}_{bw}^\alpha) \geq f(\mathcal{T}_{bw}^\alpha) - g(s)$ so if $f(\mathcal{T}_{bw}^\alpha) > f(\mathcal{T}_{fw}^{\Pi})$ in that case $h_{\mathcal{T}^\alpha}(s) = h^\alpha(s)$. Note that the proof of case 1 does not rely on $f(s) < f(r)$ and the inequalities are also true if we assume $f(s) \leq f(r)$ instead. □

Perimeter Bidirectional Abstractions

Perimeter abstractions must also be redefined to handle bidirectional and heuristic searches appropriately. First, in bidirectional search we have forward and backward perimeters. This can be leveraged in the initialization of the abstract searches by ignoring states in both closed lists. The abstract bidirectional search $\mathcal{T}^\alpha$ is initialized with the perimeter of $\mathcal{T}^{\Pi}$ as: $open(\mathcal{T}_u^\alpha)[g] = \{s_j^\alpha \mid \exists_{s\in S}\ \alpha(s) = s_j^\alpha \wedge s \in open(\mathcal{T}_u^{\Pi})[g]\}$ and $closed(\mathcal{T}_u^\alpha) = \{s_j^\alpha \mid \forall_{s\in S}\ \alpha(s) = s_j^\alpha \implies s \in closed(\mathcal{T}_{fw}^{\Pi}) \cup closed(\mathcal{T}_{bw}^{\Pi})\}$.

Second, the perimeter search is carried out with A* instead of uniform-cost search. Hence, the perimeter is not uniform, i. e., it has not only expanded all states up to a fixed radius. The resulting heuristic is still admissible as long as every state in the perimeter is closed with its optimal value, $g^*(s)$ in $\mathcal{T}_{fw}^{\Pi}$ and $h^*(s)$ in $\mathcal{T}_{bw}^{\Pi}$. However, non-uniform perimeters may cause the heuristic to be inconsistent. For example, consider two states s, t such that $s \xrightarrow{a} t$, $s \in closed(\mathcal{T}_{bw}^{\Pi})$ and $t \notin closed(\mathcal{T}_{bw}^{\Pi})$. In this case, $h(s) = h^*(s)$ and $h(t) = h^\alpha(t)$, so consistency may be violated: $h(s) \not\leq h(t) + c(a)$. However, this poses no problem if nipping is used to eliminate all states in the opposite perimeter because the heuristic will never be evaluated on those states anyway. Theorem 3 proves that the perimeter abstraction heuristic is still admissible and consistent for all relevant states, i. e., those that are not pruned by nipping.

Theorem 3. *Let $\mathcal{T}_u^\alpha$ be a perimeter abstraction in Θ^α initialized with the perimeters of $\mathcal{T}^{\Pi}$. Then, h is admissible and consistent for any state $s \notin closed(\mathcal{T}_{fw}^{\Pi}) \cup closed(\mathcal{T}_{bw}^{\Pi})$.*

Proof. A heuristic h is consistent if and only if $h(s) \leq h(t) + c(a)\ \forall s \xrightarrow{a} t$. In our case, we only contemplate states $s, t \notin closed(\mathcal{T}_{fw}^{\Pi}) \cup closed(\mathcal{T}_{bw}^{\Pi})$, so $\alpha(s)$ and $\alpha(t)$ will not be introduced in $closed(\mathcal{T}_u^\alpha)$ when initializing the abstract search. $\alpha(t)$ may be generated by $\mathcal{T}_u^\alpha$ or not. If not, $h(t) = \infty$ and consistency follows. If $\alpha(t)$ is expanded by $\mathcal{T}_u^\alpha$, $\alpha(s)$ will be inserted in $open(\mathcal{T}_u^\alpha)$ and later expanded. Hence, by consistency of h^α, it follows that $h(s) \leq h(t) + c(a)$. □

Therefore, bidirectional A* guided with perimeter abstraction heuristics returns the optimal solution since the heuristic is consistent for every state outside the perimeter and nipping prevents its evaluation in states in the perimeter.

Search Selection Strategy

The core of SymBA*, as outlined in Algorithm 1, is how to decide which search should be pushed forward at each iteration. A search is a valid candidate if and only if it is both *feasible* and *useful*. A search is *feasible* if the estimated time and number of nodes needed to perform the next step does not surpass any of the bounds imposed as parameters. The search in the original search space is always *useful*. A search in an abstract search space is *useful* if and only if it can further inform the next layer in the original search space, i.e., it has the potential of simplifying the search in the original state space by changing which states are selected for expansion.

Definition 2 (Useful abstract search)**.** *Let $\mathcal{T}_u^{\Pi}$ be an A* search over Θ^{Π} and let $\mathcal{T}_{\neg u}^\alpha$ be an abstract search over Θ^α in the opposite direction. Let S_f be the set of states currently selected for expansion in $\mathcal{T}_u^{\Pi}$, i. e., a subset of those with*

minimal f-value according to any given tie-breaking criteria. We say that $\mathcal{T}^{\alpha}_{\neg u}$ is useful for $\mathcal{T}^{\Pi}_{u}$ if $f(\mathcal{T}^{\alpha}_{\neg u}) \leq f(\mathcal{T}^{\Pi}_{u})$ and $\exists_{s \in S_f} h^{\alpha}(s)$ is not known or $B(s, \mathcal{T}^{\alpha}_{\neg u}) < h^{\alpha}(s)$.

Theorem 4. *Let $\mathcal{T}^{\Pi}_{u}$ be an A^* search informed with a heuristic generated by an abstract search $\mathcal{T}^{\alpha}_{\neg u}$. If $\mathcal{T}^{\alpha}_{\neg u}$ is not useful for $\mathcal{T}_u$, continuing the abstract search cannot alter the set of states selected for expansion, S_f.*

Proof. Continuing the abstract search can only increase the h-value of the states. To show that, consider the definition of $h_{\mathcal{T}}(s)$. On one hand, $f(\mathcal{T}^{\alpha}_{\neg u})$ and $g(\mathcal{T}^{\alpha}_{\neg u})$ monotonically increase as the search is performed, so does $B(s, \mathcal{T}^{\alpha})$. On the other hand, if h^{α} is known, its value is fixed so the only point where $h_{\mathcal{T}^{\alpha}}$ could decrease is when $\alpha(s)$ is expanded.

However, just before the expansion of $\alpha(s)$, $f(\mathcal{T}^{\alpha}_{\neg u}) - g(s)$ cannot be greater than $h^{\alpha}(s)$:

$$f(\mathcal{T}^{\alpha}_{\neg u}) - g(s) > g^*_{bw}(\alpha(s))$$
$$g^*_{bw}(\alpha(s)) + h_{bw}(\alpha(s)) - g(s) > g^*_{bw}(\alpha(s))$$
$$g^*(s) \leq g(s) < h_{bw}(\alpha(s)) \leq h^*_{bw}(\alpha(s)) \leq g^*(s)$$

Reaching a contradiction.

The only way to alter S_f is to increase the heuristic value of some $s \in S_f$. Since $h^{\alpha}(s) \leq B(s, \mathcal{T}^{\alpha}_{\neg u})$, $h(s)$ can only be increased if $h(s) < h^{\alpha}(s)$, which cannot be true if the search is not useful:

- If $f(\mathcal{T}_u) = f(s) < f(\mathcal{T}^{\alpha}_{\neg u})$ then $h_{\mathcal{T}^{\alpha}_{\neg u}}(s) < B(s, \mathcal{T}^{\alpha})$ and $h(s) = h^{\alpha}(s)$.
- If $h^{\alpha}(s)$ is known and $h^{\alpha}(s) \leq B(s, T^{\alpha})$ then $h(s) = h^{\alpha}(s)$ □

The search in the original state space is preferred whenever it is feasible. Otherwise, among all the abstract searches that are *valid candidates*, we prefer those that have a greater minimum f-value, just because they are closer to proving that the current solution is optimal. In case of a tie, the search whose next step is expected to take less time is selected.

In summary, in order to prove optimality, it is enough to expand abstract searches until $f(\mathcal{T}^{\alpha}) \geq h^*(\mathcal{I})$. All the states whose abstract counterparts have not been expanded in any of the abstract searches do not need to be explored because their f-value is guaranteed to be non-optimal.

Experiments

SymBA* is implemented on top of Fast Downward (Helmert 2006) and uses h^2 in a precomputation step to remove irrelevant actions and obtain mutex constraints for pruning the symbolic search (Alcázar and Torralba 2015). For our experiments we used the version of SymBA* that was submitted to IPC'14 after fixing some bugs. We run experiments on the optimal-track STRIPS planning instances from IPC'98 until IPC'14. All experiments were conducted on a cluster of Intel E5-2660 machines running at 2.20 GHz, with time (memory) cut-offs of 30 minutes (4 GB).

We consider a search feasible if the frontier has less than 10 million BDD nodes and each step takes less than 45 seconds, which are adequate values for the memory and time limits of our experiments. SymBA* can use any abstraction function that can be efficiently represented as BDDs such as PDBs and M&S with linear merge strategies (Edelkamp, Kissmann, and Torralba 2012; Helmert, Röger, and Sievers 2015). In our evaluation we focus on the simpler variant, PDBs. To select the "pattern" of the PDBs we follow a strategy previously used for symbolic perimeter abstractions (Torralba, Linares López, and Borrajo 2013), which selects a variable ordering and relax variables one by one until the search can be continued. We use six different variable orderings. *lev* and *rev* use the variable ordering of the BDD and its reverse, respectively. *rnd* is a completely random ordering. *cgl* and *cgr* preserve variables interconnected in the causal graph giving preference to goal variables and breaking ties by *lev* or *rnd*, respectively. Finally, *gcl* preserves goal variables preferring those that are interconnected in the causal graph and breaking ties by *lev*. We also include the two configurations of SymBA* used in IPC'14, which use a combination of abstraction strategies in a round-robin schema. *ipc*1 uses *cgr*, *gcl* and *rev*. *ipc*2 uses the same PDB strategies plus a M&S strategy based on bisimulation with a limit of 10 000 abstract states. Finally, $\emptyset$ is a strategy that stops the algorithm instead of using any abstraction, to understand in which cases abstractions are being used. As an ablation study, we run all configurations disabling the perimeter abstractions ($\neg P$) and substituting the bidirectional search in the abstract state spaces for a standard backward search ($\neg B$).

	SB	SymBA*											*Metis*
		PDB						*ipc*1	*ipc*2			$\emptyset$	
		cgl	*rev*	*lev*	*rnd*	*gcl*	*cgr*			$\neg P$	$\neg B$		
Airport(50)	27	27	27	27	27	27	27	27	27	27	27	27	**29**
Barman(34)	**18**	17	17	17	17	17	17	17	17	17	17	**18**	11
Blocks(35)	31	31	31	31	31	**32**	**32**	31	31	30	31	30	28
Childsnk(20)	4	4	4	4	4	4	4	4	4	4	4	4	**6**
Depot(22)	7	7	7	7	7	7	7	7	7	7	7	7	**9**
Driverlog(20)	12	**14**	13	13	**14**	13	**14**	**14**	**14**	**14**	**14**	12	**14**
Elevators(50)	**44**	**44**	**44**	**44**	**44**	**44**	**44**	43	43	**44**	**44**	43	40
Floortile(40)	**34**	**34**	**34**	**34**	**34**	**34**	**34**	**34**	**34**	**34**	**34**	**34**	16
FreeCell(80)	23	22	21	23	23	**26**	25	25	23	21	25	21	15
GED(20)	**20**	19	19	19	19	19	19	19	19	19	**20**	19	15
Grid(5)	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	2	2
Hiking(20)	15	15	15	15	19	18	18	**20**	**20**	18	**20**	15	14
Logistics(63)	23	25	25	25	25	25	24	25	25	25	25	23	**27**
Miconic(150)	112	107	108	108	109	108	108	108	108	108	108	108	**144**
Mprime(35)	24	23	**25**	24	**25**	**25**	**25**	**25**	24	**25**	24	23	24
Mystery(30)	15	15	15	15	15	15	15	15	15	15	15	15	**18**
NoMyste(20)	14	14	14	**17**	14	16	14	15	15	14	**17**	14	**17**
Openstk(100)	**90**	**90**	**90**	**90**	**90**	**90**	**90**	**90**	89	**90**	89	89	53
ParcPrint(50)	37	37	37	37	37	37	37	37	37	37	37	37	**50**
Parking(40)	6	4	4	4	4	4	4	3	2	1	3	1	**8**
PegSol(50)	48	48	48	48	**50**	48	49	48	48	48	**50**	48	48
PipesNT(50)	15	15	15	15	15	15	15	15	15	15	15	15	**21**
PipesT(50)	**17**	16	16	16	16	16	16	16	16	16	16	16	**17**
Rovers(40)	**14**	**14**	13	**14**	**14**	**14**	13	**14**	13	**14**	**14**	12	10
Satellite(36)	9	10	9	9	10	9	9	9	9	9	10	9	**16**
Scanlz(50)	21	21	21	21	21	21	21	21	21	21	21	21	**31**
Sokoban(50)	48	48	48	48	48	48	48	48	48	48	48	48	**50**
Tetris(17)	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**	9	**10**	9	8
Tidybot(40)	25	20	20	20	20	20	20	17	**27**	**27**	**27**	17	23
TPP(30)	**9**	8	**9**	**9**	8	**9**	8	8	8	8	8	8	8
Transport(70)	**33**	31	31	31	31	31	31	31	31	31	31	**33**	24
VisitAll(40)	18	18	18	18	18	18	**19**	**19**	**19**	18	18	18	18
Woodwrk(50)	45	45	45	45	45	45	44	43	43	43	43	43	**48**
Zenotrvl(20)	10	11	12	11	11	12	12	12	12	12	11	10	**13**
Total(1607)	968	954	955	959	965	967	963	960	964	959	**973**	936	962
Score(40)	20.93	20.78	20.81	20.93	21.09	21.17	21.06	21.11	21.24	21.00	**21.44**	20.04	19.99

Table 1: Coverage of SymBA* with different abstraction strategies, compared with SB and Metis.

Table 1 compares SymBA* with different abstrac-

tion strategies against the current state-of-the-art planner in symbolic search, bidirectional uniform-cost search (SB), and one state-of-the-art explicit-state search planner, METIS (Alkhazraji et al. 2014). Domains in which all planners got the same coverage are excluded from the table. We report total coverage and a final score that gives the same weight to every domain (versions of the same domain in different competitions are considered the same domain), normalizing the coverage of every planner by the number of problems in that domain. The results show that the use of abstractions can improve the results of our baseline, SB, in 25 problems from 14 different domains. However, generating abstractions implies a non-negligible overhead that affects negatively the coverage in 35 problems from 20 different domains. The conclusion is that using abstraction heuristics is beneficial in domains where the “right” abstraction is selected. Since the performance of all the strategies is close to a random selection of variables (*rnd*), all configurations are nearly tied in total coverage. The best strategy is *ipc*2, which won the optimal-track of IPC’14. With it, the standard configuration of SymBA* gets a score of 21.24; more than the baseline and METIS.

In the 936 problems solved by $\emptyset$, all the configurations behave in the same way. In the rest of the problems, abstractions are used in around 500 cases. This reflects that, by limiting the frontier size to 10 million BDD nodes, SymBA* is able to identify in which cases the blind search is going to fail and resort to abstractions. If this parameter is set to 1 million nodes, most configurations improve in domains where they are better than the baseline (e.g., *ipc*1/2 solve 18 problems in NoMystery except for $\neg B$ and *ipc*2 with $\neg B$ solves 11 instances in Satellite.) but their total coverage slightly decreases: $\emptyset$ solves 898 instances, *ipc*2 961 and *ipc*2 with $\neg B$, 962.

The ablation study shows that using perimeter abstractions is usually helpful. Regarding bidirectional search in abstract state spaces, however, the results are less clear. When a single PDB is used, both versions are closely tied though sometimes working best in different domains. However, when combining several strategies, as in the *ipc* configurations, the version disabling bidirectional search in abstract state spaces obtains better results. Our best configuration is *ipc*2 with backward search in the abstract state spaces, improving the previous state-of-the-art techniques both in coverage and score.

Related Work

A connection can be made with hierarchical heuristic search (HHS) algorithms (Holte, Grajkowski, and Tanner 2005). Like SymBA*, they do not precompute the abstract distance before starting the search, but do so on-demand, i.e., the abstract distance for a given abstract state is computed lazily. This allows HHS algorithms to use more informed abstractions since they do not need to traverse the entire abstract state space. The difference among the HHS algorithms lies in how the abstract distances are computed when needed. The algorithms that are closer to SymBA* in this regard are Switchback (Larsen et al. 2010) and its improved version Short-Circuit (Leighton, Ruml, and Holte 2011). They compute the abstract distances in the usual way, performing an abstract search in the opposite direction to the search that they want to inform. However, they stop the search as soon as the heuristic value for every state in the open list is known, avoiding the exploration of the entire abstract state space.

Despite the similarities between our work and these algorithms, there are noticeable differences. First of all, unlike HHS algorithms, SymBA* uses bidirectional search so it can be considered as a bidirectional hierarchical algorithm. Furthermore, we consider the inclusion of partial and perimeter abstractions which are, as discussed in the paper, needed in order to apply these ideas to planning domains. For example, Switchback and Short-Circuit do not use partial abstractions so the abstract search must continue until the heuristic value of every state in the open list is known. Therefore, in the presence of recognized dead-end states the algorithm will traverse the entire abstract state space, losing all the advantages of the algorithm. The theory presented in this paper could help to develop a version of these algorithms that can be used in planning domains with dead-end states.

Conclusions

This paper addresses the question of whether heuristics can be used to further improve the results of symbolic bidirectional uniform-cost search (SB). This is a hard task, given that multiple BHS algorithms have been proposed in the past failing to outperform A* search and SB. We have introduced a new algorithm, SymBA*, that uses bidirectional A* with abstraction heuristics. SymBA* leverages the performance of SB by deferring the use of heuristics until a blind search seems unfeasible. In order to generate heuristics for both search frontiers, a bidirectional search is carried out in an abstract state space, initialized with the current frontier as a perimeter abstraction. To this end, we extended the definition of partial and perimeter abstractions to the bidirectional case. These extensions are not limited to SymBA* and they can be applied to other algorithms that use A* to explore abstract state spaces such as the hierarchical heuristic search algorithm Switchback (Larsen et al. 2010; Leighton, Ruml, and Holte 2011).

Our experimental results show that abstractions can further improve the current state-of-the-art in symbolic bidirectional search, helping SymBA* to win the optimal track of the last IPC. However, finding the right abstractions in a domain-independent way is not a trivial task and there is still room for improvement in future work.

Acknowledgments

We’d like to thank Rosa Moreno Morales for her advice and support. This work was partially supported by MICINN projects TIN2014-55637-C2-1-R and TIN2011-27652-C03-02.

References

Alcázar, V., and Torralba, Á. 2015. A reminder about the importance of computing and exploiting invariants in plan-

ning. In *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Alcázar, V.; Fernández, S.; and Borrajo, D. 2014. Analyzing the impact of partial states on duplicate detection and collision of frontiers. In *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*, 350–354.

Alkhazraji, Y.; Katz, M.; Matmüller, R.; Pommerening, F.; Shleyfman, A.; and Wehrle, M. 2014. Metis: Arming fast downward with pruning and incremental computation. In *International Planning Competition (IPC)*, 88–92.

Anderson, K.; Holte, R.; and Schaeffer, J. 2007. Partial pattern databases. In *Proc. of the Symposium on Abstraction, Reformulation and Approximation (SARA)*, 20–34.

Barker, J. K., and Korf, R. E. 2015. Limitations of front-to-end bidirectional heuristic search. In *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)*, 1086–1092.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* 35(8):677–691.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

de Champeaux, D. 1983. Bidirectional heuristic search again. *Journal of the ACM* 30(1):22–32.

Dillenburg, J. F., and Nelson, P. C. 1994. Perimeter search. *Artificial Intelligence Journal* 65(1):165–178.

Edelkamp, S., and Kissmann, P. 2008a. Limits and possibilities of BDDs in state space search. In *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)*, 1452–1453.

Edelkamp, S., and Kissmann, P. 2008b. Partial symbolic pattern databases for optimal sequential planning. In *Proc. of the German Conference on Artificial Intelligence (KI)*, 193–200.

Edelkamp, S.; Kissmann, P.; and Torralba, Á. 2012. Symbolic A* search with pattern databases and the merge-and-shrink abstraction. In *Proc. of the European Conference on Artificial Intelligence (ECAI)*, 306–311.

Edelkamp, S. 2001. Planning with pattern databases. In *Proc. of the European Conference on Planning (ECP)*, 13–34.

Eyerich, P., and Helmert, M. 2013. Stronger abstraction heuristics through perimeter search. In *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*, 303–307.

Felner, A., and Ofek, N. 2007. Combining perimeter search and pattern database abstractions. In *Proc. of the Symposium on Abstraction, Reformulation and Approximation (SARA)*, 155–168.

Felner, A.; Zahavi, U.; Holte, R.; Schaeffer, J.; Sturtevant, N. R.; and Zhang, Z. 2011. Inconsistent heuristics in theory and practice. *Artificial Intelligence Journal* 175(9-10):1570–1603.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the ACM* 61(3):16:1–16:63.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*, 176–183.

Helmert, M.; Röger, G.; and Sievers, S. 2015. On the expressive power of non-linear merge-and-shrink representations. In *Proc. of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Holte, R. C.; Felner, A.; Sharon, G.; and Sturtevant, N. R. 2016. Bidirectional search that is guaranteed to meet in the middle. In *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)*.

Holte, R. C.; Grajkowski, J.; and Tanner, B. 2005. Hierarchical heuristic search revisited. In *Proc. of the Symposium on Abstraction, Reformulation and Approximation (SARA)*, 121–133.

Kaindl, H., and Kainz, G. 1997. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research (JAIR)* 7:283–317.

Kissmann, P. 2012. *Symbolic Search in Planning and General Game Playing*. Ph.D. Dissertation, Universität Bremen.

Kwa, J. B. H. 1989. BS*: An admissible bidirectional staged heuristic search algorithm. *Artificial Intelligence Journal* 38(1):95–109.

Larsen, B. J.; Burns, E.; Ruml, W.; and Holte, R. 2010. Searching without a heuristic: Efficient use of abstraction. In *Proc. of the AAAI Conference on Artificial Intelligence (AAAI)*, 114–120.

Leighton, M. J.; Ruml, W.; and Holte, R. C. 2011. Faster optimal and suboptimal hierarchical search. In *Proc. of the Symposium on Combinatorial Search (SoCS)*, 92–99.

McMillan, K. L. 1993. *Symbolic model checking*. Kluwer Academic publishers.

Nilsson, N. J. 1982. *Principles of Artificial Intelligence*. Springer.

Pohl, I. 1969. *Bi-directional and heuristic search in path problems*. Ph.D. Dissertation, Department of Computer Science, Stanford University.

Torralba, Á., and Alcázar, V. 2013. Constrained symbolic search: On mutexes, BDD minimization and more. In *Proc. of the Symposium on Combinatorial Search (SoCS)*, 175–183.

Torralba, Á.; Linares López, C.; and Borrajo, D. 2013. Symbolic merge-and-shrink for cost-optimal planning. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2394–2400.

Blind Search for Atari-like Online Planning Revisited

Alexander Shleyfman and **Alexander Tuisov** and **Carmel Domshlak**
Faculty of Industrial Eng. and Management
Technion, Israel
{alesh@tx, squel@campus, dcarmel@ie}.technion.ac.il

Abstract

Similarly to the classical AI planning, the Atari 2600 games supported in the Arcade Learning Environment all feature a fully observable (RAM) state and actions that have deterministic effect. At the same time, the problems in ALE are given only implicitly, via a simulator, *a priori* precluding exploiting most of the modern classical planning techniques. Despite that, Lipovetzky *et al.* (2015) recently showed how online planning for Atari-like problems can be effectively addressed using IW(i), a blind state-space search algorithm that employs a certain form of structural similarity-based pruning. We show that the effectiveness of the blind state-space search for Atari-like online planning can be pushed even further by focusing the search using both structural state similarity and the relative myopic value of the states. We also show that the planning effectiveness can be further improved by considering online planning for the Atari games as a multiarmed bandit style competition between the various actions available at the state planned for, and not purely as a classical planning style action sequence optimization problem.

Introduction

Since its introduction in 2013, the Arcade Learning Environment (ALE) draws a growing interest as a testbed for general, domain-independent planners and learners through a convenient interface to numerous Atari 2600 games (Bellemare *et al.* 2013). These games all feature a fully observable state and actions that have deterministic effect. At the same time, both the action dynamics and the reward structure of the games are given only implicitly, via a simulator, bringing the ALE setup closer to the challenges of real-world applications.

ALE supports two settings of action selection problem: an *online planning* setting where each action selection is based on a relatively short, simulated lookahead, and a *learning* setting that must produce reactive controllers for mapping states into actions after a single long session of interactions with the simulator. In this work we consider the online planning setting.

The implicit, simulator-based problem representation in ALE precludes automatic derivation of heuristic functions and other inferences developed in the scope of classical planning (Ghallab *et al.* 2004; Geffner and Bonet 2013). This state of affairs restricts the algorithmic choices to blind (aka not future estimating) search algorithms, such as blind best-first search and Monte-Carlo tree search algorithms. The first results on domain-independent online planning in ALE have been reported by Bellemare et al. (2013), where the search algorithm of choice was the popular Monte-Carlo tree search algorithm UCT (Kocsis and Szepesvári 2006). In particular, UCT was shown there to substantially outperform breadth-first search (BrFS). The latter result probably came at no surprise since blind best-first search methods such as BrFS are inherently ineffective over large state spaces. Recently, however, Lipovetzky et al. (2015) showed that this is not the end of the story for breadth-first search. In particular, they showed that IW(i), a pruning-enhanced successor of BrFS originated in work in classical planning (Lipovetzky and Geffner 2012), favorably competes with UCT on the Atari games.

In this work we show that the effectiveness of blind state-space search for deterministic online planning in Atari-like problems can be pushed even further by focusing the search using both structural state similarity and the relative myopic value of the states. We introduce and evaluate *prioritized IW*(i), a simple extension of IW(i) that approximates breadth-first search with duplicate detection and state reopening, and show that it very favorably competes with IW(i) on the Atari games. We then revisit the basic objective underlying deterministic online planning. We argue that the effectiveness of online planning for the Atari games and related problems can be further improved by considering this problem as a multiarmed bandit style competition between the actions available at the state planned for (and not purely as a classical planning style action sequence optimization problem). Following this lead, we introduce a simple modification of prioritized IW(i) that fits the modified objective, and empirically demonstrate the prospects of this direction.

Background

The Atari 2600 games exposed by ALE represent a broad range of problems that are characterized by means of a finite set of states, with each state being represented by a complete assignment to some n finite domain variables $X_1, \ldots, X_n$, an initial state s_0, a finite set of actions, a transition function $s' = f(a, s)$ where s' is the state resulting from applying action a in state s, and real-valued rewards $r(a, s)$ that result from applying a in s. The transition function and rewards in ALE are implemented by a game simulator and

thus are not known to the planner *a priori*. At the same time, the environment is fully observable: when applying action a is simulated in state s, the resulting new state $f(a, s)$ and the collected reward $r(a, s)$ are revealed to the planner. The state of the game is simply captured by the content of Atari's RAM of 128 bytes. Different choices of factoring this RAM into a set of state variables are possible, and, as it is typically the case with feature generation in learning, this choice of factoring may have a substantial impact on the planner's performance. This aspect of the problem is tangential to our focus here; for ease of comparability, we adopt the previous work's factoring along 128 variables, each representing the value of the respective memory byte, and thus having the domain of 256 values (Bellemare *et al.* 2013; Lipovetzky *et al.* 2015).

IW(i), an algorithm that has recently been shown by Lipovetzky et al. (2015) to exhibit state-of-the-art performance on the Atari games, is a regular breadth-first search with the following modification: When a state s is generated, it is assigned a *novelty* penalty, and is pruned if the penalty is higher than i. The novelty penalty of a newly generated state s is 1 if s is the first state generated in the search that makes true some atom $X = x$, else it is 2 if s is the first state that makes a pair of atoms $X = x \wedge Y = y$ true, and so on. If the problem state is represented by n atoms, then IW(n) simply corresponds to breadth-first search with duplicate detection, while values of i lower than n induce breadth-first searches with respectively more aggressive state pruning.

In classical planning, the primary termination condition for the search process is the achievement of the goal. In problems with a more general and/or unknown reward structure, such as the Atari games, the termination is determined by a search resource budget, such as a time window or a limit on the number of generated nodes. The accumulated reward $R(s)$ of a generated state s is $R(s) = R(s') + r(a, s)$ where s' is the unique parent state of s. The best path is then a state-space path from the initial state to a state that maximizes the accumulated reward. In online planning, the first action along this path is then executed, the system transitions to a new state, and the search process starts over from that new initial state. In what follows, the state that is planned for at the current iteration is denoted by s_0.

Prioritized Pruning

While the experiments of Lipovetzky et al. (2015) showed that IW(1) performs on the Atari games at the level of UCT, a closer look at the results suggests that the strength of these two algorithms is somewhat complementary: Out of the 54 games used in the experiments, IW(1) scored more than 150% of the UCT's score in 17 games, while UCT scored more than 150% of the IW(1)'s score in 13 games. The two algorithms are too different to easily characterize the problems on which each has an *a priori* advantage, and yet one key difference between them immediately suggests itself as a natural explanation of the complementarity of the two: While the exploration of UCT is biased towards the regions of the state-space that already appear rewarding, the exploration of IW(1) has no such a bias whatsoever.

As a variant of blind search that aims at combining both strengths, Lipovetzky et al. (2015) evaluated 2BFS, a best-first search algorithm with two queues: one queue ordered in increasing order of the novelty penalty, and a second queue ordered in a decreasing order of the accumulated reward. In one iteration, the best first search picks up the best node from one queue, and in the second iteration it picks up the best node from the other queue, with all the generated nodes being added to both queues. This way, while IW(i) progresses in a breadth-first manner while pruning states based on their novelty, 2BFS progresses (in its first queue) in the best-first manner while only prioritizing the states based on their novelty.

In that respect, an important property of most of the Atari games is that the state of the game changes not only due to the actions of the player but also due to the change of the environment—the cars keep moving, the rocks keep falling, etc.—while the changes that the player can make to the state are rather limited.[1] As a result, a state reachable in k steps from the initial state is likely to be novel with respect to the states reachable in less than k states. Thus, if the best-first search selects the state to expand based on its relative novelty, then at least some of its children are also likely to exhibit relatively high novelty, ending up high in the queue. Such a chain effect makes the search much more *depth-first*, and, since the state-space is typically much larger than what the search can explore within a reasonable search budget, the explored region of the state-space is likely to have a narrow focus. In turn, this biases action selection at s_0 towards action sequences that collect rewards far from s_0 while possibly missing alternatives that bring the rewards earlier as well. Interestingly, the second queue of 2BFS, prioritized by the states' reward-so-far, does not necessarily balance this phenomenon: Since the first rewarding state is more likely to be found within the tunnel created by the depth-first-like progress of the first queue, the second queue is likely to join expanding that tunnel, possibly making it wider, but still, abandoning the exploration of the state-space under the *myopically* less appealing alternatives.

At first view, the breadth-first searching IW(1) is expected to behave differently. However, in our experiments we consistently observed IW(1) exhibiting a very similar "single tunnel" phenomenon. This seems to happen precisely because of the aforementioned structure of the Atari games due to which *the likelihood of the states of the same shallowness to survive the novelty pruning decays very rapidly with the position of the states in the queue.* Indeed, in the experiments of Lipovetzky et al. (2015), neither 2BFS exhibited a substantial advantage over IW(1) nor the other way around, leaving open the quest for an effective interplay between the novelty and the reward-so-far promise.

We now show that plugging a bias towards the reward-so-

[1] This is very different from the typical structure of the benchmarks used in the classical planning research where the set of actions controlled by the planner is rather rich, but at the same time, these actions are responsible for most, if not all, the changes made to the state. At least in part, the latter can be attributed to the fact that encoding environment changes in the PDDL language is not an easy task.

far *into* the actual state pruning mechanism offers a promising direction for addressing this quest: Even if done in a rather simple manner as in the *prioritized IW*(i) procedure described below, this approach results in an algorithm that strongly outperforms IW(1) and UCT. Prioritized IW(i), or p-IW(i), for short, deviates from IW(i) twofold:

1. While preserving the breadth-first dynamics of IW(i), the ties in the queue are broken to favor states with higher accumulated reward.
2. Every i-set of atoms $\mathbf{x}$ is schematically pre-assigned a "reward" of $\widehat{r}(\mathbf{x}) = -\infty$. Given that, a generated state s is considered novel if, for some i-set of atoms $\mathbf{x}$ in s, we have $R(s) > \widehat{r}(\mathbf{x})$. If that holds, then (and only then) s is not pruned, and, for each i-set of atoms $\mathbf{x}$ in s, we update its reward to $\widehat{r}(\mathbf{x}) := \max\{R(s), \widehat{r}(\mathbf{x})\}$.

The two modifications of p-IW(i) with respect to IW(i) bring the reasoning about the reward accumulated by the states directly into the mechanism of state pruning, addressing two complementary questions—what states should lead the pruning, and what states should be pruned—as follows.

First, the regular breadth-first search is driven by two principles: always expand one of the shallowest states in the queue and never put a state into the queue twice. The later *duplicate pruning* makes BrFS a graph search rather than a tree search, leading to up to exponential savings in the search effort while preserving completeness. In that respect, IW(i) is BrFS in which state duplication is *over-approximated* by state (non-)novelty: If a search node generated by BrFS is pruned due to its duplication, then, for any i, that search node would be pruned by IW(i), but not necessarily vice versa (unless $i = n$). In classical planning, this "non-novel as duplicate" over-approximation has a strong semantics via a notion of "problem width" (Lipovetzky and Geffner 2012). In the settings of ALE, however, this over-approximation is motivated only informally, by a similarity to the novelty-based search methods developed independently in the context of genetic algorithms (Lehman and Stanley 2011). In the latter methods, individuals in the population are not ranked according to the optimization function but in terms of how much they are different from the rest of the population, thus encouraging global exploration rather than (greedy) search for local improvements.

Though better exploration is indeed what online planning effectiveness boils down to (Bubeck *et al.* 2011), the direct linkage to the diversity-driven genetic algorithms has an important weakness. Suppose that the two shallowest states in the search queue of IW(i) are s_1 and s_2, and suppose further that the children of s_1 make the children of s_2 non-novel and vice versa. In other words, expanding any of these two states blocks the search under the other state. In IW(i), the choice between s_1 and s_2 remains arbitrary. However, if the accumulated reward of s_1 is higher than s_2, then, *ceteris paribus*, it is only reasonable to assume that the best extension of s_1 is more rewarding than the best extension of s_2, and thus s_1 should better be expanded before s_2. This example emphasizes the difference between the evolutionary search in genetic algorithms and state-space forward search: While the former typically examines fully specified candidates to the problem solution, the latter gradually expands *partial solutions* in the form of path prefixes. Under the additive structure of the accumulated reward, the quality of partial solutions lower bounds the quality of their extensions, making total ignorance of the accumulated reward of the states in the queue rather questionable. The first modification of p-IW(i) with respect to IW(i) approaches precisely this issue under the conservative, *ceteris paribus* semantics, preserving the breadth-first search dynamics of the search.

Suppose now that IW(i) generates a state s such that $R(s) > R(s')$ for all the previously generated states s'. Despite the fact that the extensions of the respective path to s are now the most promising candidates for the best solution that IW(i) can possibly compute from now on, if s is evaluated as non-novel, then it is pruned, independently of its accumulated reward. The second deviation of prioritized IW(i) from IW(i) takes the accumulated reward of the generated state s into account in the actual decision whether s should be pruned or not. Specifically, a newly generated state s in prioritized IW(i) is pruned if, for every i-set of atoms $\mathbf{x}$ in s, there was a previously generated state s' that contains $\mathbf{x}$ and has $R(s') \geq R(s)$.

Similarly to the way the state pruning in IW(i) can be understood as an over-approximation of the standard duplicate pruning, the state pruning in p-IW(i) can be understood as an *over-approximation of duplicate pruning with state reopening*. In BrFS, if the solution optimality is of interest, then, if a previously generated state s is rediscovered through a different path with a higher accumulated reward, then s is "reopened", either by re-starting the search from s onwards, or by propagating the new accumulated reward of s to its descendants in the queue. In that respect, if a state generated by BrFS with node reopening is pruned, then, for any i, that search node would be pruned by p-IW(i), but not necessarily vice versa. In particular, this modification allows for a substantial alleviation of the "single tunnel" phenomenon exhibited by IW(i), keeping the search wider but only when the extended search breadth is justified by the accumulated reward of the respective states.

Experiments – Plane

We tested p-IW(1) and IW(1) on 53 of the 55 different games considered by Bellemare et al. (2013): The SKIING game was already left out in the experiments of Lipovetzky et al. (2015) due to certain issues with the reward structure of this game. We decided to also leave out BOXING because, in the single player setting of ALE, scoring in this game boils down to striking in arbitrary directions since the second player is doing nothing, and therefore every algorithm will trivially score the possible maximum.

We used the implementation of IW(1) by Lipovetzky et al. (2015), and have implemented p-IW(1) on top of it. To focus the comparison on the effectiveness of individual online decisions, both algorithms have been evaluated in a memory-less setting. This is in contrast to the experiments of Lipovetzky *et al.* (2015) in which IW(1) reused the frames in the sub-tree of the previous lookahead that is rooted in the selected child, to allow for a direct comparison with the results reported for UCT by Bellemare *et*

	150K		10K	
Game	p-IW(1)	IW(1)	p-IW(1)	IW(1)
ALIEN	**4939**	4705	**1638**	1473
AMIDAR	**1186**	938	67	**78**
ASSAULT	**1700**	591	**423**	373
ASTERIX	**172413**	30780	5985	**6683**
ASTEROIDS	**63520**	29884	2192	**2224**
ATLANTIS	**151720**	52453	**144850**	126703
BANK HEIST	**296**	**296**	**67**	63
BATTLE ZONE	**7767**	5000	**2900**	2133
BEAM RIDER	**4487**	3398	2445	**2730**
BERZERK	**854**	639	**208**	200
BOWLING	**27**	**27**	**28**	27
BREAKOUT	**291**	224	**400**	344
CARNIVAL	**2773**	2509	**2141**	1832
CENTIPEDE	**163917**	59913	**140171**	134542
CHOPPER COMMAND	**5653**	2040	**2230**	2157
CRAZY CLIMBER	**107673**	37350	**114157**	37013
DEMON ATTACK	**24153**	12448	4845	**6098**
DOUBLE DUNK	**-6**	**-6**	**-14**	-18
ELEVATOR ACTION	**8910**	4217	**2597**	2057
ENDURO	420	**432**	**0**	**0**
FISHING DERBY	**-8**	-9	**-82**	-83
FREEWAY	**30**	**30**	**23**	**23**
FROSTBITE	**353**	199	257	**259**
GOPHER	9756	**11852**	9546	**15019**
GRAVITAR	**2943**	2270	**343**	315
HERO	4969	**5483**	2159	**2170**
ICE HOCKEY	**43**	41	**-7**	**-7**
JAMESBOND	**173**	152	**32**	**32**
JOURNEY ESCAPE	7973	**8560**	440	**1293**
KANGAROO	1057	**1130**	587	**753**
KRULL	**10293**	4332	**4464**	3475
KUNG FU MASTER	**67163**	33903	**26610**	26250
MONTEZUMA REVENGE	**0**	**0**	**0**	**0**
MS PACMAN	**11451**	8219	3511	**3835**
NAME THIS GAME	**11302**	6087	**12445**	11004
PONG	**14**	13	**-20**	**-20**
POOYAN	**2252**	1312	1945	**2271**
PRIVATE EYE	**72**	0	**93**	20
QBERT	**1640**	1249	1441	**1527**
RIVERRAID	**8707**	4055	**3303**	3095
ROAD RUNNER	**80900**	39133	**0**	**0**
ROBOTANK	**59**	57	**3**	2
SEAQUEST	**19007**	2747	245	**260**
SPACE INVADERS	**2037**	1151	**227**	211
STAR GUNNER	**14193**	2783	1097	**1190**
TENNIS	**10**	9	**-24**	**-24**
TIME PILOT	**31767**	5903	**18797**	15140
TUTANKHAM	136	**140**	**182**	147
UP N DOWN	**93305**	75088	**2717**	2576
VENTURE	**240**	150	**0**	**0**
VIDEO PINBALL	**413976**	223772	**286921**	237078
WIZARD OF WOR	**111487**	88953	**6373**	4270
ZAXXON	**15247**	9200	**0**	**0**
# times best (53 games)	**47**	11	**38**	24
# times 10K better than 150K			6	13

Table 1: Performance of p-IW(1) and IW(1) in 53 Atari 2600 games. The algorithms are evaluated over 30 runs for each game. The maximum episode duration is 18000 frames, with the lookahead per decision being limited to 150000 simulated frames in columns 2-3 and to 10000 simulated frames in columns 4-5. Per lookahead budget, the average scores in bold show best performer and the summary of the performance is given at the bottom of the table.

al. (2013) under a similar setting. Following Lipovetzky *et al.* (2015), each action selection decision was given a lookahead budget of 150000 simulated frames (or, equivalently, 30000 search nodes), the lookahead depth was limited to 1500 frames, and the accumulated rewards were discounted as $R(s') = R(s) + \gamma^{d(s)+1} r(s, a)$ where s is the unique parent of s', a is the respective action, d is the distance from the root node, and the discount factor[2] was set to $\gamma = 0.995$. To reduce the variance, each game was played 30 times, with the reported results being averaged across these runs.

Columns 2-3 in Table 1 shows that p-IW(1) rather consistently outperforms IW(1). Out of the 53 games, p-IW(1) achieved higher average scores in 42 games, 5 games ended up with a draw, and IW(1) achieved higher average scores in 6 games. Of the latter, the highest achievement of IW(1) was the 22% score difference in GOPHER, while p-IW(1) outscored IW(1) by more than 50% on 25 games. In fact, comparing our results with the results reported by Lipovetzky *et al.* (2015) for IW(1) *with* memory, p-IW(1) without memory scored higher than IW(1) with memory on 14 games.

In general, we have noticed that both p-IW(1) and IW(1) typically did not use the entire budget of 150000 simulated frames per decision. To examine the score improvement as a function of budget, we have also tested them under a budget of only 10000 simulated frames per decision, all else being equal. The results are shown in columns 4-5 of Table 1. As one would expect, the scores here are typically lower than these achieved under the 150000 frames budget, and, since p-IW(1) brings an approximation of state reopening, typically it benefits of the budget increase much more than IW(1). More interestingly, while the higher budget "misled" p-IW(1) on 6 games, in none of these cases the score loss was substantial. In contrast, IW(1) did worse with 150000 frames budget than with 10000 frames budget on 13 games, with the loss being substantial on 6 games, namely ATLANTIS, CENTIPEDE, GOPHER, NAME THIS GAME, POOYAN, and TIME PILOT.

Experiments – Frame Reuse

Another evaluation step we used was the testing of p-IW(1) in the setting described in in Lipovetzky *et al.* (2015). As reported in those two works, all of the algorithms reuse the frames in the sub-tree of the previous lookahead that is rooted in the selected child, deleting its siblings and their descendants. More precisely, no calls to the emulator are done for transitions that are cached in that sub-tree, and such reused frames are not discounted from the budget that is thus a bound on the number of new frames per lookahead. In addition, in IW(1) and p-IW(1), the states that are reused from the previous searches are ignored in the computation of the novelty of new states so that more states can escape pruning.

As in the previous subsection we evaluated the performance over the 53 Atari 2600 games. The algorithms, 2BFS,

[2] The discount factor results in a preference for rewards that can be reached earlier, which is a reasonable heuristic given the budget limits of the lookahead search. At the same time, choosing the right discount factor is a matter of tuning.

UCT, and BrFS was evaluated over 10 runs (episodes) for each game, the algorithms, IW(1), p-IW(1) was evaluated over 30 runs (episodes) for each game. This was done to reduce the standard deviation values for the two algorithms in question. As before the maximum episode duration was 18000 frames and every algorithm was limited to a lookahead budget of 150,000 simulated frames. Figures for UCT, and BrFS taken from Bellemare *et al.* (2013), whether the figures for 2BFS are taken form Lipovetzky *et al.* (2015). Numbers in bold show best performer in terms of average score. The IPPC-like score in Table 2 was calculated with the BrFS algorithm serving as a baseline, using the following formula for an algorithm $\psi \in \{\text{IW}(1), \text{p-IW}(1), \text{2BFS}, \text{UCT}\}$:

$$\frac{score(\psi) - score(\text{BrFS})}{max_{x \in \{\text{IW}(1),\text{p-IW}(1),\text{2BFS},\text{UCT}\}} score(x) - score(\text{BrFS})}$$

if $score(\psi) > score(\text{BrFS})$, and 0 otherwise.

Table 2 shows that p-IW(1) rather consistently outperforms all other evaluated algorithms. Out of the 53 games, p-IW(1) achieved higher average scores in 34 games, and tying up on BOWLING with IW(1), on TENNIS with IW(1) and 2BFS, and finally on PONG with all algorithms except BrFS. On the other hand, IW(1) was best in 4 games, and 2BFS and UCT in 6. In the terms of IPPC-like score, p-IW(1) achieves 144% of the cumulative score of IW(1), 144% of the 2BFS score, and 172% of this of UCT.

IW(1) and p-IW(1) with memory mostly outperform IW(1) and p-IW(1) without it, with a couple of exceptions being PRIVATE EYE and STAR GUNNER. This may happen due to the inheritance of the search tree without updating the novelty table. Both IW(1) and p-IW(1) with memory get ”broad” tree from the previous step, and the expansion of all leaves in these trees limits the further advancement of the search. Spending most of the budget on the ”breadth” nodes and thus, reducing the depth of the search tree. Another exception is the CRAZY CLIMBER game, where IW(1) with memory mostly outperform IW(1) without it. Here the case is slightly different, since IW(1) without memory and a budget of 10000 frames outperforms the same algorithm with a budget of 150000 frames. We assume this happens because of the density of the rewards in the game. Since the IW(1) is guided only by the novelty measure, and not by the reward at hand, the recommendation results in a choice not of the ”best“ action, but of the action with the longest rollout.

Racing Blind Search

Approximating state duplication by state non-novelty allows IW(i) to search deeper in the state-space, possibly reaching rewarding states that lie far from the initial state. At the same time, this specific approximation often results in a highly unbalanced exploration of the state space. p-IW(i) partly alleviates the latter phenomenon, but the extent to which this is achieved depends on the reward structure of the specific game. Recall that the original objective pursued by the, both heuristically guided and blind, best-first forward search procedures is to compute a sequence of actions from the initial state to a state that maximizes the accumulated reward, even

Game	IW(1)	p-IW(1)	2BFS	UCT	BrFS
ALIEN	28238	**38951**	12252	7785	784
AMIDAR	1775	**3122**	1090	180	5
ASSAULT	896	**1970**	827	1512	414
ASTERIX	145067	**319667**	77200	290700	2136
ASTEROIDS	52170	**68345**	22168	4661	3127
ATLANTIS	150327	**198510**	154180	193858	30460
BANK HEIST	601	**1171**	362	498	22
BATTLE ZONE	7667	9433	**330880**	70333	6313
BEAM RIDER	9851	**12243**	9298	6625	694
BERZERK	**1915**	1212	802	554	195
BOWLING	**69**	**69**	50	25	26
BREAKOUT	401	477	**772**	364	1
CARNIVAL	5898	**6251**	5516	5132	950
CENTIPEDE	98922	**193799**	94236	110422	125123
CHOPPER COMMAND	12310	**34097**	27220	34019	1827
CRAZY CLIMBER	36010	**141840**	36940	98172	37110
DEMON ATTACK	20177	**34405**	16025	28159	443
DOUBLE DUNK	0	8	21	**24**	-19
ELEVATOR ACTION	13097	16687	10820	**18100**	730
ENDURO	**499**	497	359	286	1
FISHING DERBY	22	**42**	6	38	-92
FREEWAY	31	**32**	23	0	0
FROSTBITE	2040	**6427**	2672	271	137
GOPHER	18175	**26297**	15808	20560	1019
GRAVITAR	4517	**6520**	5980	2850	395
HERO	12769	**15280**	11524	1860	1324
ICE HOCKEY	55	**62**	49	39	-9
JAMESBOND	**20668**	15822	10080	330	25
JOURNEY ESCAPE	42263	**65100**	40600	12860	1327
KANGAROO	**8243**	5507	5320	1990	90
KRULL	6357	**15788**	4884	5037	3089
KUNG FU MASTER	63570	**86290**	42180	48855	12127
MONTEZUMA REVENGE	13	27	**500**	0	0
MS PACMAN	22869	**30785**	18927	22336	1709
NAME THIS GAME	9244	14118	8304	**15410**	5699
PONG	**21**	**21**	**21**	**21**	-21
POOYAN	10460	15832	10760	**17763**	910
PRIVATE EYE	-60	21	**2544**	100	58
QBERT	5139	**44876**	11680	17343	133
RIVERRAID	6865	14437	8304	**15410**	2179
ROAD RUNNER	85677	**120923**	68500	3875	245
ROBOTANK	67	**75**	52	50	2
SEAQUEST	13972	**35009**	6138	5132	288
SPACE INVADERS	2812	3076	**3974**	2718	112
STAR GUNNER	1603	1753	**4660**	1207	1345
TENNIS	**24**	**24**	**24**	3	-24
TIME PILOT	35810	**65213**	36180	63855	4064
TUTANKHAM	167	158	204	**226**	64
UP N DOWN	104847	**120200**	54820	74474	746
VENTURE	1107	**1167**	980	0	0
VIDEO PINBALL	288394	**471859**	62075	254748	55567
WIZARD OF WOR	122020	**161640**	81500	105500	3309
ZAXXON	33840	**39687**	15680	22610	0
# times best (53 games)	7	**37**	8	7	0
IPPC-like score	32.17	**46.37**	31.83	26.83	0

Table 2: Performance of p-IW(1) vs. IW(1), 2BFS, UCT, and BrFS. The experimental setup is similar to this in Lipovetzky *et al.* (2015), with the lookahead budget of 150000 frames.

if not in absolute terms but only best effort. In the context of this objective, it is actually hard to argue whether a more balanced exploration of the state space is more rational than a less balanced exploration, and if so, what kind of balance we should strive for here. In fact, in the absence of any extra knowledge about the problem, expanding an already rewarding sequence of actions is arguably more rational than searching elsewhere.

In the context of online planning, however, computing an as rewarding as possible sequence of actions is *not* the actual objective of the planner. Let $A(s_0) = \{a_1, \ldots, a_k\}$ be the actions applicable at the current state s_0 and, for $1 \leq l \leq k$, let π_l be the most rewarding action sequence applicable in s_0 that starts with a_l. The actual objective in online planning is not to find the most rewarding action sequence π_{l^*} among $\pi_1, \ldots, \pi_k$ but only to find the index l^* of that sequence, that is, to find the identity of the first action along π_{l^*}. At least in theory, the latter objective is less ambitious than the former since computing π_{l^*} implies finding l^* but obviously not the other way around. In practice, this difference in objectives suggests that various adaptations can be found beneficial when transferring the techniques from the classical, open-loop AI planning to the closed-loop online planning.

To exemplify the prospects of such adaptation, consider the following example. Whether we apply prioritized or regular IW(i) (or, for that matter, any other blind forward search procedure), suppose that, at a certain stage of the search process, the states in the queue all happen to be descendants of the same action a_l applicable at the planned state s_0. If one of these states has the maximum accumulated reward among all the states generated so far, then the search can be terminated right away: No matter how much further we will continue searching, a_l will remain the action of our choice at s_0, that is, l will remain our estimate for the desired action index l^*. Furthermore, let $Q_1, \ldots, Q_k$ be a cover of the search queue Q of either prioritized or regular IW(i), such that, for $1 \leq l \leq k$, $s \in Q_l$ if one of the most rewarding action sequences generated so far from s_0 to s starts with a_l. Given that, the candidates for a_{l^*} can be restricted to a subset A of $A(s_0)$ if $\{Q_l \mid a_l \in A\}$ induces a set cover of Q.

In sum, the search procedures in the context of online planning should aim at the *competition* between the actions in $A(s_0)$. UCT and BrFS actually appear to be more faithful with this objective than both IW(i) and p-IW(i) yet not without caveats:

- The UCT algorithm is grounded in the UCB (upper-confidence bound) Monte-Carlo procedure for optimizing online action selection in multiarmed bandits (MABs) (Auer *et al.* 2002). However, UCT has at least two substantial shortcomings in the settings of online planning for the Atari games: First, while the UCB procedure is optimized for the learning-while-acting settings of MAB, the simple uniform and round-robin sampling of the actions provide much better formal and empirical guarantees when it comes to online action selection with MAB simulators (Bubeck *et al.* 2011). Thus, within the scope of the Monte-Carlo tree search algorithms, the more balanced sampling algorithms such as BRUE (Feldman and Domshlak 2014) are *a priori* more appropriate. Second, the usage of the upper-confidence bounds and of the very averaging Monte-Carlo updates in UCT aims at estimating the mean value of the policies under stochasticity of the action outcomes. Since all the actions in the Atari games are deterministic, neither of these tools is semantically meaningful here and actually harm the convergence of the decision process.
- In contrast, BrFS is built for deterministic actions and by definition runs a fair competition between the actions in $A(s_0)$ in terms of the search horizon. The absence of selectiveness, however, makes BrFS uncompetitive in problems with large search width such as the Atari games where all the actions are applicable in every state.

Combining the selectiveness of p-IW(i) with a uniformly balanced exploration of the actions, we have evaluated the following simple modification of p-IW(1), referred to in what follows as *racing p-IW*(1):

1. Expand the initial state s_0. For every subset of actions $A \subseteq A(s_0)$ that result in the same successor of s_0, eliminate from $A(s_0)$ all but one action $a \in A$ that maximizes $r(s_0, a)$.
2. Let the (pre-pruned as above) action set $A(s_0)$ be $\{a_1, \ldots, a_k\}$. At iteration m, restrict the selection from the search queue only to successors of $f(s_0, a_{m\%k})$.
3. At every stage of the search, if the search queue contains only successors of $f(s_0, a)$ for some action $a \in A(s_0)$ and a state in the queue corresponds to the most rewarding path generated so far, terminate the search and recommend (aka execute) a.

Note that nothing in the above modification is specific to p-IW(1), and thus one can adapt any forward state-space search algorithm to the action selection objective of online planning in exactly the same manner.

Table 3 compares the performance of racing p-IW(1) with that of p-IW(1) and IW(1). The experimental setup remains as before, with the lookahead budget of 150000 frames. While p-IW(1) was still the leader with best performance in 34 games, racing p-IW(1) achieved the best average scores in 16 games, including some very substantial leads such as in BATTLE ZONE, GOPHER, POOYAN, and QBERT. While racing p-IW(1) was the only algorithm to score in the very challenging game MONTEZUMA REVENGE, at the moment we have no evidence that this should be attributed to anything but pure chance. At the same time, the seemingly small advantage of racing p-IW(1) over p-IW(1) and IW(1) in FREEWAY is actually substantial since the maximum score in this game is 38, and this canonical Atari game posed a challenge to both BrFS and UCT, with IW(1) being the first algorithm to score in this game at all (Lipovetzky *et al.* 2015).

Summary and Future Work

Online planning with simulators provide a challenging testbed for action planning since most of the sophisticated techniques for scaling up planning systems rely upon inference over propositional encodings of actions and goals that

are “hidden” by the simulator. Previous work showed that a blind forward search algorithm IW(1) achieves state-of-the-art performance in such planning problems around the Atari video games, with the key to success being structural, similarity-based approximation of duplicate pruning (Lipovetzky *et al.* 2015).

We have shown that the effectiveness of blind state-space search on deterministic online planning like in the Atari games can be further improved by (a) combining approximated duplicate pruning with an approximate state reopening, and (b) reshaping the dynamics of the forward search algorithms to better fit the objective of selecting and executing only a single action at a time. Our experiments show that modifying IW(i) along these two lines results in algorithms, p-IW(1) and racing p-IW(1), that both substantially outperform IW(1) on the Atari games.

The simple concept of the racing search algorithms for deterministic online planning suggests numerous directions for future investigation. First, while the state pruning in our racing p-IW(i) was done based on a global view on state novelty, whether/when this globality is friend or foe is yet to be investigated: On the one hand, it is not hard to verify that the globally reasoned duplicate pruning will always improve the efficiency of the racing search at least as much as any locally reasoned duplicate pruning. On the other hand, pruning a state in one branch based on its similarity (but not equivalence!) to a state in another branch is not necessarily the best thing to do. As another issue, if the number of actions examined for the current state does not decrease for a substantial chunk of the lookahead budget, then it seems natural to consider a mechanism for gradual “candidate rejection”, possibly in the spirit of algorithms for budgeted pure exploration in stochastic multiarmed bandit problems like Sequential Halving (Karnin *et al.* 2013).

References

P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2-3):235–256, 2002.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An evaluation platform for general agents. *JAIR*, 47:253–279, 2013.

S. Bubeck, R. Munos, and G. Stoltz. Pure exploration in finitely-armed and continuous-armed bandits. *Theor. Comp. Sci.*, 412(19):1832–1852, 2011.

Z. Feldman and C. Domshlak. Simple regret optimization in online planning for Markov decision processes. *JAIR*, 51:165–205, 2014.

H. Geffner and B. Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool, 2013.

M. Ghallab, D. Nau, and P. Traverso. *Automated Planning*. Morgan Kaufmann, 2004.

Z. S. Karnin, T. Koren, and O. Somekh. Almost optimal exploration in multi-armed bandits. In *ICML*, pages 1238–1246, 2013.

Game	p-IW(1)	IW(1)	R. p-IW(1)
ALIEN	4939	4705	**4995**
AMIDAR	**1186**	938	911
ASSAULT	**1700**	591	572
ASTERIX	**172413**	30780	83983
ASTEROIDS	**63520**	29884	7780
ATLANTIS	**151720**	52453	133943
BANK HEIST	296	296	**303**
BATTLE ZONE	7767	5000	**11600**
BEAM RIDER	**4487**	3398	4127
BERZERK	**854**	639	496
BOWLING	27	27	**36**
BREAKOUT	291	224	**455**
CARNIVAL	2773	2509	**4270**
CENTIPEDE	**163917**	59913	72441
CHOPPER COMMAND	**5653**	2040	3433
CRAZY CLIMBER	**107673**	37350	57693
DEMON ATTACK	**24153**	12448	13927
DOUBLE DUNK	-6	-6	**-5**
ELEVATOR ACTION	**8910**	4217	7010
ENDURO	420	**432**	218
FISHING DERBY	-8	-9	**0**
FREEWAY	30	30	**32**
FROSTBITE	**353**	199	238
GOPHER	9756	11852	**16707**
GRAVITAR	**2943**	2270	1423
HERO	4969	**5483**	2922
ICE HOCKEY	**43**	41	17
JAMESBOND	173	152	**183**
JOURNEY ESCAPE	7973	**8560**	3303
KANGAROO	1057	1130	**3323**
KRULL	**10293**	4332	5692
KUNG FU MASTER	**67163**	33903	31050
MONTEZUMA REVENGE	0	0	**17**
MS PACMAN	**11451**	8219	8861
NAME THIS GAME	**11302**	6087	7957
PONG	**14**	13	12
POOYAN	2252	1312	**11116**
PRIVATE EYE	**72**	0	-1
QBERT	1640	1249	**8838**
RIVERRAID	**8707**	4055	3880
ROAD RUNNER	**80900**	39133	40547
ROBOTANK	**59**	57	26
SEAQUEST	**19007**	2747	1112
SPACE INVADERS	**2037**	1151	1365
STAR GUNNER	**14193**	2783	1293
TENNIS	**10**	9	**10**
TIME PILOT	**31767**	5903	6643
TUTANKHAM	136	140	**144**
UP N DOWN	**93305**	75088	34605
VENTURE	**240**	150	53
VIDEO PINBALL	**413976**	223772	202279
WIZARD OF WOR	**111487**	88953	70257
ZAXXON	**15247**	9200	2607
# times best (53 games)	**34**	3	16
# times better than IW(1) w/memory	14	4	7

Table 3: Performance of *racing* p-IW(1) vs. p-IW(1) and IW(1). The experimental setup is similar to this in Table 1, with the lookahead budget of 150000 frames.

L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, pages 282–293, 2006.

J. Lehman and K. O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evol. Comp.*, 19(2):189–223, 2011.

N. Lipovetzky and H. Geffner. Width and serialization of classical planning problems. pages 540–545, 2012.

N. Lipovetzky, M. Ramírez, and H. Geffner. Classical planning with simulators: Results on the Atari video games. In *IJCAI*, pages 1610–1616, 2015.

Delete-free Reachability Analysis for Temporal and Hierarchical Planning

Arthur Bit-Monnot
LAAS-CNRS, Université de Toulouse
Toulouse, France
arthur.bit-monnot@laas.fr

David E. Smith and **Minh Do**
NASA Ames Research Center
Moffet Field, CA, USA
{david.smith, minh.do}@nasa.gov

Abstract

Reachability analysis is a crucial part of the heuristic computation for many state of the art classical and temporal planners. In this paper, we study the difficulty that arises in assessing the reachability of actions in planning problems containing sets of interdependent actions, notably including problems with required concurrency as well as hierarchical planning problems. In temporal planners, this complication has been addressed by augmenting a delete-free relaxation with additional relaxations, but this can result in weak pruning of the search space. To overcome this problem, we describe a more sophisticated method for reachability analysis that uses Dijkstra's algorithm for propagation of times through a reachability graph, combined with a pruning mechanism that recognizes unachievable cycles.

We also extend our approach to handle hierarchical planning problems, in which an action and its subactions are naturally interdependent. Evaluations were conducted on a diverse set of temporal domains using FAPE, a constraint-based temporal and hierarchical planner.

1 Introduction

Reachability analysis is crucial in computing heuristics guiding many classical and temporal planners. This is typically done by relaxing the action delete lists and constructing the reachability graph. This graph is then used as a basis to extract a relaxed plan, which serves as a non-admissible heuristic estimate of the actual plan reaching the goals from the current state. Due to the relaxation, the reachability analysis is optimistic and can also provide a lower-bound on the actual "cost" of reaching the goals.

Temporal planning poses some additional challenges for reachability analysis due to the temporal objective function of minimizing the plan's makespan. This objective function leads to the requirements on the heuristic guidance to not only estimate the total cost but also the earliest time at which goals can be achieved. This can be accomplished on the reachability graph by labeling: (1) propositions by the minimum time of the effects that can achieve them; and (2) actions by the maximum time of the propositions they require as conditions. Since the reachability graph construction process progresses as time increases, when all *start* conditions are reachable, a given action a is eligible to be added to the graph. However, there is the additional problem that

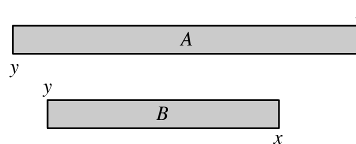


Figure 1: Two actions: A with a start effect y and an end condition x, and B with a start condition y and an end effect x.

a's end conditions must also be reachable, although they do not need to be reachable until the end time of a. To see why this is a problem for the conventional way of building the reachability graph, consider the two actions in Figure 1: action B achieves the end condition for action A, but requires a start effect of A before it can start. Thus, B cannot start before A, but A cannot end until after B has ended. This means that A is not fully reachable until B is reachable, but B is not reachable unless A is reachable. Whether this turns out to be possible depends on whether B fits inside of A. In this example, the reasoning is simple enough, but more generally, B might be a complex chain of actions.

Planners such as COLIN (Coles et al. 2012) and POPF (Coles et al. 2010) address this problem by splitting durative actions into instantaneous start and end events, and forcing a time delay between the start and end events. In our example, the start of A would be reachable, leading to the start of B being reachable, which leads to the end of B being reachable, and finally the end of A being reachable. This approach therefore concludes that A is reachable. Unfortunately, the same conclusion is reached in COLIN or POPF even when B does not fit inside of A, because this "action-splitting" approach allows A to "stretch" beyond its actual duration. While this is a satisfactory relaxation for problems with few interdependencies, it does not provide very good heuristic guidance for problems that involve a lot of action nesting, such as hierarchical style container actions that expand into subactions.

In this paper, we first present an approach to reachability analysis for durative actions that addresses the nesting problem described above. We use Dijkstra's algorithm for propagation of times through a reachability graph, together with a pruning mechanism that recognizes unachievable cycles. In the second part, we study some of the difficulties

that arise when performing reachability analysis in hierarchical planning and show that those challenges are similar to the ones encountered in temporal planning. We devise a compilation procedure that exposes the hierarchical features as additional conditions and effects in durative actions. The compiled problem is used as an input for reachability analysis of hierarchical planning problems.

2 Preliminaries

We first describe the temporal action model and other basic elements used throughout the rest of the paper.

2.1 Temporal Planning Model

In PDDL 2.2, a planning problem P is represented by a tuple $P \doteq \langle V, I, T, G, A \rangle$ where:

- V is a set of propositions.
- I is the initial state: a complete set of assignments of value T or F to all propositions in V.
- T is a set of timed-initial-literals, which are tuples $\langle [t]\ f := v \rangle$ with $f \in V$ and $t \in \mathbb{R}^+$ is the wall-clock time at which f will be assigned the Boolean value v.
- $G \subseteq V$ is a goal state: a set of propositions that need to be true when the plan finishes executing.
- A is a set of durative actions, each of the form $a \doteq \langle D_a, C_a, E_a \rangle$ where:
  - D_a is a set of constraints on the duration of the action. The actual duration of an action is referred to as d_a and takes a value in R^+ that is consistent with D_a.
  - C_a is the set of conditions. Each $p \in C_a$ is of the form $\langle (st_p, et_p)\ f = \mathtt{T} \rangle$ where st_p and et_p indicate the start and end time of the condition p relative to the action's start time. When $st_p = et_p = 0$ or $st_p = et_p = d_a$ then p is an instantaneous *at-start* or *at-end* condition. When $st_p = 0$ and $et_p = d_a$ then p is an *overall* durative condition. $f \in V$ is a proposition that must be true over the specified time period.
  - E_a is a set of instantaneous effects, each $e \in E_a$ is of the form $\langle [t_e]\ f := v \rangle$ where $t_e \doteq 0$ or $t_e \doteq d_a$ is the relative time at which the *at-start* or *at-end* effect e occurs.

A plan π of P is a set of tuples $\langle t_a, a, d_a \rangle$, in which an action $a \in A$ is associated to a wall-clock start time t_a and a duration d_a that satisfies the constraints in D_a. π is valid if it is executable in I and achieves all goals in G.

Beyond PDDL 2.2: We extend the temporal action model in PDDL2.2 to allow conditions expressed over sub-intervals of actions, and effects at arbitrary time points during an action. These features turn out to be particularly useful for encoding many temporal planning applications. We do this by allowing the times st_p and et_p of a condition p or t_e of an effect e to take an arbitrary value in $[0, d_a]$.

Discrete time model: Unlike PDDL 2.2, which assumes the continuous time model, we assume the discrete time model in which time changes in discrete steps. This is not essential to our approach, but simplifies the presentation. We therefore represent the durative conditions $\langle (st_p, et_p)\ f = \mathtt{T} \rangle$ in PDDL 2.2 as a sequence of consecutive instantaneous conditions $\langle [t]\ f = \mathtt{T} \rangle$ with $st_p \leq t \leq et_p$. For the rest of this paper, we will assume that all action conditions and effects occur at discrete time-steps t specified as either $t = \delta$ (at a constant duration δ after the start time of action a) or $t = d_a - \delta$ (at a constant duration δ before the end time of action a).

2.2 Delete-free Elementary Actions

To estimate when each fact can be achieved, our reachability analysis utilizes *elementary actions*, which are artificial actions created from the original temporal actions defined in the domain description. Elementary actions contain: (1) only a single 'add' effect and (2) the minimal set of conditions required to achieve that effect. Specifically, given a temporal action a, the set of elementary actions for a is created by:

1. Removing all 'delete' effects of a.
2. For each 'add' effect $e = \langle [t_e]\ f := \mathtt{T} \rangle$, creating a new elementary action a_e with e as the only effect of a_e.
3. Adding each condition $p \in C_a$ to a_e with an *optimistic* timing constraint on when p is needed. By *optimistic*, we mean requiring each $p \in C_a$ as late as possible with respect to the time at which e is achieved. Let lb_{d_a} and ub_{d_a} be the lower and upper bounds on the duration d_a of a and d_{a_e} be the duration of a_e, then this maximum lateness can be achieved by fixing the value of d_{a_e}:
   - If $t_e = \delta$, then set $d_{a_e} = ub_{d_a}$
   - If $t_e = d_a - \delta'$, then set $d_{a_e} = lb_{d_a}$

Figure 2 shows an example of a move action for a rover and its two elementary actions: $a_1 = moveFree(r, l, l')$ represents the start effect $e_1 = \langle [1]\ free(l) := \mathtt{T} \rangle$ and $a_2 = moveAt(r, l, l')$ represents the end effect $e_2 = \langle [d]\, at(l') := \mathtt{T} \rangle$. Since $t_{e_1} = 1$, we fix the duration of $moveFree(r, l, l')$ to be the upper-bound value of the duration d of the original action $move(r, l, l')$ and thus $d_{a_1} = 50$. On the other hand, d_{a_2} is set to be the lower-bound value 40 of d.

For easier illustration, an elementary action a is represented graphically by an action node a with (1) an outgoing effect edge $a \xrightarrow{X} f$ representing its effect $\langle [X]\, f := v \rangle$; and (2) one incoming condition edge $f \xrightarrow{-Y} a$ for each condition $\langle [Y]\, f = \mathtt{T} \rangle \in C_a$. The *reachability graph* is a pair $\langle N, E \rangle$ with N a set of action and fluent nodes and E a set of condition and effect edges for all elementary actions. The edges from fluents to actions encode the necessary delay between the conditions of an action and the action. The semantics is different for edges from actions to fluents, as each edge $a \xrightarrow{X} f$ represents one possible action choice a for achieving the fluent f.

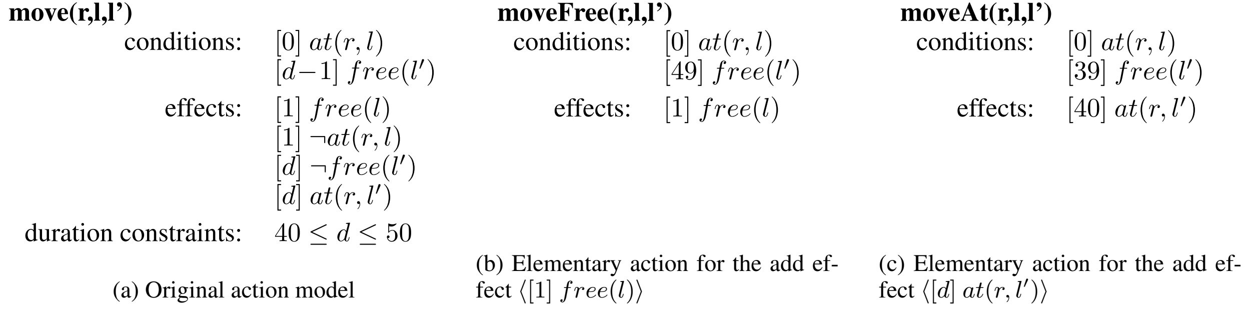


Figure 2: Action to move a rover r from location l to l' and its two elementary actions. The rover frees its original location one time unit after departing and requires its target location to be free one time unit before arriving.

3 Reachability Analysis

3.1 Definitions

An elementary action a is *applicable* once all of its preconditions are met. An action with an effect f is called an *achiever* of f. A fluent f becomes *achievable* after one of its achievers a becomes applicable, with the achievable time depending on the starting time of a and the time constraint on the effect of a that enables f. As a consequence of using the delete-free elementary actions, once a fluent is achievable at time t or an action is applicable at time t, it stays achievable/applicable at all subsequent time points.

Action a is applicable at time t (denoted by $applicable(a,t)$) if for all conditions $p = \langle[X]f = \texttt{T}\rangle \in C_a$, p is achievable at time $t_p = t + X$ (denoted by $achievable(p, t_p)$). Similarly, a fact f is achievable at time t (i.e., $achievable(f,t)$) if there exist one achiever a of f such that a has an effect $e = \langle[X]\ f := v\rangle$ and a is applicable at time $t_a = t - X$ (i.e., $applicable(a, t_a)$).

We define as the *earliest appearance* of action a (denoted by $ea(a)$), the smallest t for which $applicable(a,t) = \texttt{T}$. Similarly, the *earliest appearance* of a fluent f is the smallest t for which $achievable(f,t) = \texttt{T}$. The computation of $ea(a)$ and $ea(f)$ has traditionally been done by following the dynamic programming update rules below:

Initialization:
$$\forall f \in I : ea(f) = 0$$
$$\forall f \notin I : ea(f) = \infty$$
$$\forall a \in A : ea(a) = \infty$$

Updating:
$$\forall a \in A : ea(a) = \max_{\langle[X]\ f=\texttt{T}\rangle \in C_a} ea(f) - X$$
$$\forall f \in V : ea(f) = \min_{\langle[X]\ f:=\texttt{T}\rangle \in E_a} ea(a) + X$$

When the updating rules above are properly applied repeatedly, the collective values of $ea(f)$ and $ea(a)$ will reach a fix-point. We use $ea^*(f)$ and $ea^*(a)$ to denote the final values of $ea(f)$ and $ea(a)$ for all fluents f and actions a. If $ea^*(f) < \infty$ or $ea^*(a) < \infty$, we say that f or a is *reachable*, denoted by $reachable(f) = \texttt{T}$ and $reachable(a) = \texttt{T}$.

Intuitively, if an action or a fluent is not reachable by applying elementary actions, then it can not be achieved using the original action model. Therefore, a fluent cannot be true at a time earlier than $ea^*(f)$ and a reachable action A can never be executed at a time earlier than $ea^*(A)$.

3.2 Causal Loops in Temporal Planning

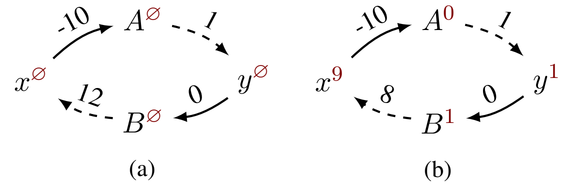


Figure 3: Two reachability graphs built from the actions of Figure 1. A is an action with a duration of 10 time units, an end condition $\langle[10]\ x = \texttt{T}\rangle$ and a start effect $\langle[1]\ y := \texttt{T}\rangle$. B has the start condition $\langle[0]\ y = \texttt{T}\rangle$ and the end effect $\langle[d_B]\ x := \texttt{T}\rangle$. The duration of B, d_B, is respectively set to 12 and 8 in graphs (a) and (b). Each node is annotated in red with its earliest appearance time or $\varnothing$ if it is not reachable.

Let us consider what would happen when applying the dynamic programming rules to the reachability graphs of Figure 3. In Figure 3a, where B does not 'fit' into A, it would behave as expected: the positive cycle would maintain all the nodes with an infinite earliest start (i.e. unreachable). On the other hand, Figure 3b depicts a self-supporting causal loop where the effect y of A allows B to produce x early enough to achieve the end condition of A. Such self-supporting causal loops can be identified as cycles of negative or zero length in the reachability graph. If the cycle is of negative length, as in Figure 3b, the earliest appearances would be infinitely updated towards $-\infty$. In the case of a zero length cycle, the dynamic programming rules fail to detect that an update is needed to mark the node as reachable.

Temporal planning problems with such causal loops are identified by Cooper, Maris, and Régnier (2013) as *temporally-cyclic* problems and are characterized by sets of interdependent actions. The difficulty in handling them has been avoided in state of the art temporal planners by ignoring any condition that might be achieved through a self-supporting causal loop. In POPF (Coles et al. 2010), the reachability model is built by splitting durative actions into an instantaneous start-action and an instantaneous end-action, with the start-action using only the 'at-start' conditions. Applied to our example in Figure 2, this would

roughly result in ignoring the $\langle [49]\, free(l') = \mathtt{T} \rangle$ condition of $moveFree(r, l, l')$. This additional relaxation leads to reachability models that disregard any condition that could lead to a self-supporting causal loop.

Self supporting causal loops always contain an *after-condition* : a condition in C_a that appears at the same time or later than the effect of an elementary action a (Cooper, Maris, and Régnier 2013). To make the identification of after-conditions easier, we assume that a condition $\langle [X]\, f = \mathtt{T} \rangle$ is an after-condition iff $X > 0$. This restriction means that any negative edge in the reachability graph represents an after-condition. If necessary, this can be enforced by artificially shifting the start of all elementary actions to be one time unit before their effect.

3.3 Reachability Analysis with Causal Loops

To handle after-conditions during reachability analysis, as detailed in Algorithm 1, we alternate two steps: (1) a first step propagates achievement times while ignoring all after-conditions, performed by a Dijkstra pass on the graph limited to positive edges; then (2) a second step that enforces all after-conditions, represented by negative edges, that were ignored in the first step. Those two steps are complemented with a pruning mechanism that repeatedly detects nodes in positive cycles.

Algorithm 1 begins by selecting a set of *assumed reachable* nodes from which to start the propagation process (lines 2-10). The obvious candidates are fluents appearing in the initial state I and in timed initial literals T. We also optimistically select all actions that have no *before-conditions*, i.e., actions where every condition is an after-condition. Assumed reachable nodes are inserted into a priority queue Q of $\langle n, t \rangle$ pairs where n is a node of the reachability graph and t is a candidate time for its earliest appearance.

We then iteratively extend the initial set of assumed reachable nodes with all fluents that have an assumed reachable achiever and all actions whose every before-condition is assumed reachable. This is done by a Dijkstra-like propagation (line 13), that extracts the nodes in Q in the order of increasing appearance time. Those extracted nodes are marked as reachable and their successors are inserted into Q. The algorithm slightly differs from Dijkstra's as it ensures that an action node is enqueued only if all of its before-conditions have been already marked reachable (lines 36-38).

As a second step, we revise our optimistic assumptions by incorporating the ignored after-conditions:

- Line 16 removes from the graph any action a with an after-condition on an unreachable fluent f. More specifically, the RECURSIVELYREMOVE procedure marks its parameter as unreachable and removes it from the graph. This removal process is recursive: if a removed action a is the only achiever for a fluent f then f is removed as well (and as a consequence all actions depending on f will also be removed). Furthermore, if the first achiever of a fluent is removed from the graph and there is at least one other achiever for it, then the fluent is added back to Q with an updated earliest appearance.
- Line 18 takes an after-condition of an action a on a reachable fluent f and enforces the minimal delay δ between $ea(f)$ and $ea(a)$. If the current delay is not sufficient, a is added to Q and will be reconsidered upon the next Dijkstra pass.

Algorithm 1 Algorithm for identifying reachable nodes in a reachability graph and computing their earliest appearance.

```
 1: ⟨N, E⟩ ← Reachability Graph
 2: Q ← ∅     ▷ Priority queue of ⟨node, time⟩ ordered by
    increasing time
 3: for all n ∈ N do
 4:     reachable(n) ← F
 5:     if n is an action with no before-conditions then
 6:         Q ← Q ∪ {⟨n, 0⟩}
 7: for all ⟨[t] f := T⟩ ∈ T do            ▷ timed initial literals
 8:     Q ← Q ∪ {⟨f, t⟩}
 9: for all f := T ∈ I do                          ▷ initial state
10:     Q ← Q ∪ {⟨f, 0⟩}
11:
12: while Q non empty do
13:     DIJKSTRAPASS
14:     for all f -δ-> a ∈ after-condition edges do
15:         if ¬reachable(f) then
16:             ⟨N, E⟩ ← RECURSIVELYREMOVE(a)
17:         else if ea(a) < ea(f) + δ then
18:             Q ← Q ∪ {⟨a, ea(f) + δ⟩}
19:     for n ∈ N do
20:         if n is late then
21:             ⟨N, E⟩ ← RECURSIVELYREMOVE(n)
22:
23: procedure DIJKSTRAPASS
24:     while Q non empty do
25:         ⟨n, t⟩ ← pop(Q)
26:         if n already expanded in this pass then
27:             continue
28:         if reachable(n) ∧ ea(n) ≥ t then
29:             continue
30:         reachable(n) ← T
31:         ea(n) ← t
32:         if n is an action with an effect edge n -δ-> f then
33:             Q ← {⟨f, t + δ⟩}
34:         else
35:             for all a conditioned on n do
36:                 if all before cond. of a are reach. then
37:                     t' ← max_{f -δ-> a ∈ E} ea(f) + δ
38:                     Q ← Q ∪ {⟨a, t'⟩}
```

Finally, *late nodes* are marked unreachable and removed from the graph (line 21). We say that a node n is *late* if for any *non-late* node n', $ea(n') + d_{max} < ea(n)$ where d_{max} is the highest delay on any edge of the graph. In practice, this means that nodes are partitioned into non-late nodes and late nodes, these two sets being separated by a temporal gap of at least d_{max}. The intuition, as demonstrated in the next section, is that the earliest appearance of a late node is being pushed back due to unachievable cycles.

The two-step process is repeated to take into account the newly updated reachability information. In the subsequent runs, the Dijkstra algorithm will start propagating the updated nodes from the previous run, with lines 28-29 making sure that the earliest appearance values $ea(n)$ are never decreased to an overly optimistic value. The algorithm detects a fix-point and exits if the queue is empty, meaning that after-conditions did not trigger any change.

3.4 Analysis and Related Models

We now explore some of the characteristics of Algorithm 1. The first Dijkstra pass acts as an optimistic initialization: it identifies a set of possibly reachable nodes and assigns them earliest appearance times. All operations after this first pass will only *(i)* shrink the set of reachable nodes; and *(ii)* increase the earliest appearance times.

Proposition 3.1. *If a node n is reachable, then $ea(n)$ converges towards $ea^*(n)$. If a node n' is not reachable then $ea(n')$ either remains at ∞ or diverges towards ∞ until it is removed from the graph.*

Proof (sketch). A node n is reachable if there is either a path from initial facts to n or n is part of a self-supporting causal loop (i.e. cycle of negative or zero length). Consequently repeated propagations will eventually converge. On the other hand, an unreachable node either depends on an unreachable node or is involved only in causal cycles of strictly positive length (such as the one depicted in Figure 3a). If the node was ever assumed reachable, its earliest appearance will thus be increased until it is removed from the graph. □

Proposition 3.2. *If a node is late, then it is not reachable.*

Proof (Sketch). The intuition is that the gap between non-late and late nodes appeared because late nodes are delaying each other due to positive causal cycles. We first show that any late node delayed to its current time is due to a dependency on another late node: because the temporal gap is bigger than all edges in the graph, a non-late node could not have influenced a late node. It follows that any late node depends on at least one other late node. Furthermore a late node necessarily participates in a positive cycle or depends on a late node that does. From there, one can show that at least one node n in this group is involved only in positive cycles. Any other possibility (path from timed initial literals or negative cycle) would have resulted in n being less than d_{max} away from a non-late node. □

It follows from propositions 3.1 and 3.2 that Algorithm 1 produces a reachability model R_∞ that contains a node n and its earliest appearance $ea^*(n)$ iff n is reachable in the relaxed problem. In the worst case, computing this model has a pseudo-polynomial complexity since there may be as many as d_{max} iterations of the algorithm (d_{max} being the highest delay in the graph). The cost of each iteration is dominated by the Dijkstra pass of $O(|N| \times \log(|N|) + |E|)$.

Discussion: One might consider computing various approximations of R_∞ by limiting the number of iterations to a fixed number K, making the algorithm strongly polynomial and producing a reachability model R_K. In the special case where $K = 1$, this is equivalent to performing a single Dijkstra pass and removing all actions with an unreachable after-condition. Increasing K would allow us to better estimate the earliest appearances and detect additional late nodes.

Another simplification is to ignore all negative edges of the reachability graph, which can be done by stopping Algorithm 1 after the first Dijkstra pass. In practice, this model simply ignores after-conditions and it has all the characteristics of the temporal planning graph of POPF: (1) the separation of durative actions into at-start and at-end instantaneous actions is done by the transformation into elementary actions; (2) the minimal delay between matching at-start and at-end actions is enforced by the presence of start conditions in the elementary actions representing the end effects; and (3) any end condition appearing in the elementary action of a start effect would be ignored because it would be an after-condition. Since it is a direct adaptation of the techniques used in POPF to our more complex action representation, we call this model R^{popf}.

It is interesting to note that R^{popf} and R_∞ are equivalent on all problems with no after-conditions. Classical planning obviously falls in this category as well as any PDDL model with no at-start effect or no at-end condition. In fact, on such problems R^{popf} and R_∞ are equivalent to building a temporal planning graph, with no significant computational overhead.

4 Extending to Hierarchical Models

While hierarchical planning is a dominant approach for modeling and solving real-world planning applications, it still mostly requires manual work to model and control its search space. In planners such as SHOP2 (Nau et al. 2003), this is done by manually annotating methods with additional preconditions that are checked before introducing a method into the plan. This approach allows for early dead-end detection and has proven to be extremely efficient for solving complex problems. Nonetheless, it does have important drawbacks. First, manual annotation requires significant domain modeling efforts and can easily lead to modeling errors and incomplete domain descriptions. Second, conditions on methods can only be efficiently tested on fully defined states. For that reason, HTN planners usually restrict themselves to finding totally-ordered plans, which do not work for planning problems with required concurrency. Reachability analysis thus constitutes a critical step to improve the performance of partially-ordered hierarchical planners, which can find plans with required concurrency, and also reduce the laborious domain engineering effort for HTN planners that only find totally-ordered plans.

The main difficulty of automated reachability analysis for hierarchical problems lies in the interactions between causal and hierarchical constraints. HTN planning has three main characteristics: (1) a method must eventually have all its subtasks achieved; (2) a method or operator can only appear in a plan if it is refining an existing task of the plan; and (3) all conditions of operators and methods must be causally supported by earlier effects. While there are known tech-

niques to check (3), integrating (1) and (2) causes interdependent actions and remains a difficult issue.

In this section, we propose a transformation of operators and methods to expose those hierarchical constraints as additional conditions and effects. This allows us to perform the reachability analysis proposed in the previous section on models integrating hierarchical and causal information.

4.1 Hierarchical Model

We extend the temporal planning model from Section 2.1 to support the definition of hierarchical problems. A temporal planning problem P is extended to contain:

- a set $\mathcal{T}$ of task symbols
- a set of goal tasks $G_\mathcal{T}$. A goal task $g_\tau \in G_\mathcal{T}$ is of the form $\langle [st_\tau, et_\tau]\ \tau \rangle$ where st_τ and et_τ are timepoints taking value in R^+ and $\tau \in \mathcal{T}$ is a task symbol. The goal task g_τ states that the plan should contain an action achieving the task τ and spanning the temporal interval $[st_\tau, et_\tau]$.

Each action $a \in A$ is associated to a task symbol $\tau_a \in \mathcal{T}$, representing the task achieved by a and a set of subtasks S_a. A subtask is denoted by $\langle [st_\tau, et_\tau]\ \tau \rangle$ where τ is a task symbol and st_τ and et_τ are timepoints. The intuition is that for each subtask $\langle [st_\tau, et_\tau]\ \tau \rangle \in S_a$, a requires an action achieving τ and executing over the interval $[st_\tau, et_\tau]$. This model does not make any distinction between methods and operators as it is usually done in HTN planning. To make this distinction, one could simply partition A into actions with no effects (i.e. methods) and actions with no subtasks (i.e. operators).

In addition to the requirements of Section 2.1, we consider the following conditions for a plan π to be valid:

- for any task $g_\tau = \langle [st_\tau, et_\tau]\ \tau \rangle$ appearing in $G_\mathcal{T}$ or as a subtask of an action in π, there is an action in π achieving g_τ. An action $\langle t_a, a, d_a \rangle \in \pi$ is said to achieve g_τ if $\tau = \tau_a$, $st_\tau = t_a$ and $et_\tau = t_a + d_a$.
- for any action $\langle t_a, a, d_a \rangle \in \pi$, there is a task g_τ appearing in $G_\mathcal{T}$ or as a subtasks of an action in π such that a achieves g_τ.

The latter condition is a consequence of the search mechanism of HTN planners in which every action is introduced to fulfill a given pending task. When combined with the former, it results in interdependencies as a method both requires the presence of actions fulfilling its subtasks and enables their presence.

4.2 Flattening Transformation

We now propose a compilation of a hierarchical problem into the temporal model of Section 2.1. This flattening procedure is meant to allow a reachability analysis on causal models that retain the hierarchical constraints from the original problem.

For each task $\tau \in \mathcal{T}$ from the hierarchical model, the set of propositions V is extended with three new propositions $started(\tau)$, $ended(\tau)$ and $required(\tau)$. They respectively represent that an action achieving τ starts, finishes or is required to fulfill a pending task τ.

put-on-table-from-stack(x)

task:	put-on-table(x)
conditions:	$\varnothing$
effects:	$\varnothing$
subtasks:	$[d_1, d_2]$ unstack(x)
	$[d_3, d_4]$ put-down(x)
constraints:	$0 < d_1 < d_2 < d_3 < d_4 < d$

(a) A high level action (or method) to move a block x from the top of a stack to the table.

put-on-table-from-stack(x) (flattened model)

conditions:	$[0]$ $required$(put-on-table(x))
	$[d_1]$ $started$(unstack(x))
	$[d_2]$ $ended$(unstack(x))
	$[d_3]$ $started$(put-down(x))
	$[d_4]$ $ended$(put-down(x))
effects:	$[0]$ $started$(put-on-table(x))
	$[d_1]$ $required$(unstack(x))
	$[d_3]$ $required$(put-down(x))
	$[d]$ $ended$(put-on-table(x))
constraints:	$0 < d_1 < d_2 < d_3 < d_4 < d$

(b) The flattened action after compiling away its hierarchical properties.

Figure 4

A hierarchical action $a \in A$ is transformed into a 'flat' action a_{flat} with:

- all conditions, effects and constraints of a
- one additional condition $\langle [0]\ required(\tau_\mathrm{a}) = \mathtt{T} \rangle$
- one additional at-start effect $\langle [0]\ started(\tau_\mathrm{a}) := \mathtt{T} \rangle$ and one additional at-end effect $\langle [d_a]\ ended(\tau_\mathrm{a}) := \mathtt{T} \rangle$
- for each subtask $\langle [st_\tau, et_\tau]\ \tau \rangle$ of a:
  - two additional conditions $\langle [st_\tau]\ started(\tau) = \mathtt{T} \rangle$ and $\langle [et_\tau]\ ended(\tau) = \mathtt{T} \rangle$
  - one additional effect $\langle [st_\tau]\ required(\tau) := \mathtt{T} \rangle$

For each goal task $\langle [st_\tau, et_\tau]\ \tau \rangle \in G_\mathcal{T}$, a timed initial literal $\langle [st_\tau]\ required(\tau) := \mathtt{T} \rangle$ is added to T and a goal $ended(\tau)$ is added to G.

It is important to note that the resulting 'flat' problem is a relaxation of the original one. Indeed, a given subtask $\langle [st_\tau, et_\tau]\ \tau \rangle$ yields two conditions $\langle [st_\tau]\ started(\tau) = \mathtt{T} \rangle$ and $\langle [et_\tau]\ ended(\tau) = \mathtt{T} \rangle$. Those two conditions can be fulfilled by distinct actions, thus ignoring temporal constraints on the unique action that should have achieved the subtask in the original model. This relaxed transformation is simply meant to expose hierarchical features of the problem to reachability analysis. Actions resulting from this compilation step can be split into elementary actions and added to a reachability graph (Section 3).

An example of this transformation is given in Figure 4b. The problem has interdependencies as the presence of the *put-on-table-from-stack* method both requires and allows

the presence of its *unstack* and *put-down* subactions. Indeed, an *unstack(x)* action would have a start condition $\langle [0]\ required(\text{unstack(x)}) = \texttt{T} \rangle$ which is achieved by the method *put-on-table-from-stack*. Concurrently, this method has the condition $\langle [d_1]\ started(\text{unstack(x)}) = \texttt{T} \rangle$ which would be achieved as a start effect of the *unstack(x)* action.

5 Empirical Evaluation

We implemented our reachability analysis technique within FAPE, a partial-order temporal planner (Dvorak et al. 2014a). FAPE takes problems modeled in ANML (Smith, Frank, and Cushing 2008). ANML natively supports: (1) conditions and effects at arbitrary time points and over arbitrary intervals within an action; and (2) hierarchical structures. FAPE supports most of the features of ANML and is capable of both hierarchical and generative planning.

Like other partial-order planners, FAPE searches for a plan by fixing flaws in partial plans until no flaws remain. Every time a partial plan p is extracted from the open queue, reachability analysis is performed and provides an updated set of impossible actions and fluents. If it can be verified that from p: (1) all goals are reachable, and (2) all unrefined tasks have a possible refinement, then we expand p by fixing one of its flaws. If this is not the case, p is a dead-end and is discarded. Reachability results are also used to filter out flaw resolvers involving impossible actions or fluents.

We evaluate our reachability analysis technique on several temporal domains with and without hierarchical decomposition, the former involving many instances of required concurrency and interdependent actions. The *satellite*, *rovers*, *tms*, *logisitcs* and *hiking* domains are direct translations of the eponymous domains from the International Planning Competition (IPC) into ANML. The domain files were manually translated while the translation of problem instances was automated. The *handover* domain is a robotics problem presented in (Dvorak et al. 2014b) and the *docks* domain is the dock worker domain from (Ghallab, Nau, and Traverso 2004). Hierarchical versions of the domains have their names appended with '-hier'. All experiments were conducted on an Intel Xeon E3 with 3GB of memory and a 30 minutes timeout.

Table 1 and Figure 5 present the number of problems solved using different reachability models. R_∞ outperforms the other configurations: solving the highest number of problems on all but one domain. R_5 and R_1 are respectively second and third best performers while R^{popf} does not provide significant pruning of the search space; the computational overhead makes it perform slightly worse than no reachability checks (denoted by $\varnothing$). As expected, on temporally simple problems (non-hierarchical domains in our test set), all configurations show similar performance.

Table 2 presents the percentage of actions detected as unreachable by different configurations. As expected, R_∞, R_5, R_1 and R^{popf} perform identically on temporally simple problems. However, R^{popf} is largely outperformed on all but one hierarchical domains.The good performance of R_1 with respect to R^{popf} shows that a single iteration is often sufficient to capture most of the problematic after-conditions. However, on more complex problems such as *hiking-hier* and *docks-hier*, more iterations are beneficial both in terms of detected unreachable actions and solved problems.

	R_∞	R_5	R_1	R^{popf}	$\varnothing$
satellite (20)	14	14	14	14	**15**
satellite-hier (20)	**17**	**17**	**17**	**17**	16
rovers (40)	**25**	**25**	**25**	**25**	**25**
rovers-hier (40)	**22**	**22**	**22**	**22**	**22**
tms-hier (20)	**7**	**7**	**7**	**7**	**7**
logistics (28)	**8**	**8**	**8**	**8**	**8**
logistics-hier (28)	**28**	**28**	**28**	6	9
hiking-hier (20)	**20**	17	16	15	17
handover-hier (20)	**16**	**16**	**16**	7	7
docks-hier (18)	**17**	13	12	7	7
Total (254)	**174**	167	165	128	133

Table 1: Number of solved tasks for various domains with a 30 minutes timeout. The best result is shown in bold. The number of problem instances is given in parenthesis.

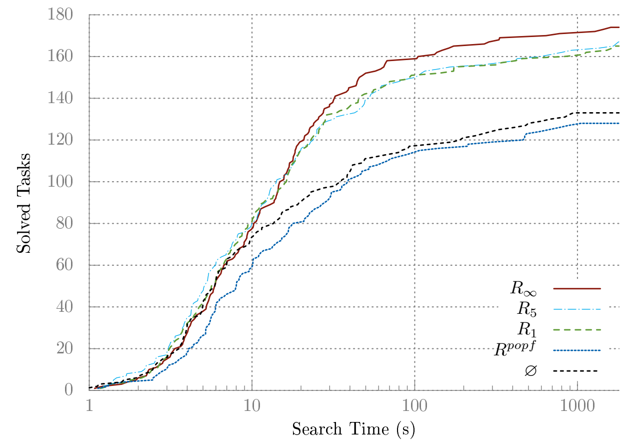


Figure 5: Number of solved tasks by each configuration within a given time amount.

	R_∞	R_5	R_1	R^{popf}	$\varnothing$
satellite	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
satellite-hier	**14.1**	**14.1**	**14.1**	**14.1**	0.0
rovers	**43.5**	**43.5**	**43.5**	**43.5**	0.0
rovers-hier	**72.6**	**72.6**	**72.6**	27.1	0.0
tms-hier	**87.9**	**87.9**	**87.9**	0.0	0.0
logistics	**34.5**	**34.5**	**34.5**	**34.5**	0.0
logistics-hier	**94.6**	**94.6**	**94.6**	15.5	0.0
hiking-hier	**38.1**	36.5	36.5	0.0	0.0
handover-hier	**99.2**	**99.2**	**99.2**	3.5	0.0
docks-hier	**85.2**	52.6	52.6	0.0	0.0

Table 2: Percentage of ground actions detected as unreachable from the initial state. For each problem instance, the percentage is obtained by comparing the number of ground actions detected as unreachable from the initial state with the original number of ground actions. Those values are then averaged over all instances of a domain.

	R_∞	R_5	R_1	R^{popf}	∅
satellite	**100** (1)	**100**	**100**	**100**	–
satellite-hier	**100** (2)	**100**	**100**	**100**	–
rovers	**100** (1)	**100**	**100**	**100**	–
rovers-hier	**100** (4)	99.2	56.7	54.2	–
tms-hier	**100** (6)	69.3	14.2	14.2	–
logistics	**100** (1)	**100**	**100**	**100**	–
logistics-hier	**100** (2)	**100**	2.8	2.8	–
hiking-hier	**100** (9)	**100**	71.7	71.7	–
handover-hier	**100** (43)	98.2	5.7	5.7	–
docks-hier	**100** (37)	73.0	29.8	29.8	–

Table 3: Average admissible makespans for different reachability models. Those are computed by taking the earliest appearance of the latest satisfied goal from the initial state, and normalizing on the value computed for R_∞. For R_∞, we also indicate the average number of iterations needed to converge on the first propagation of each instance (in parenthesis).

Table 3 presents the value that would have been taken by the admissible h^{max} heuristic with different reachability models. On all but one hierarchical model, both R_1 and R^{popf} largely underestimate the makespan of a solution. Indeed, not propagating after-conditions makes them miss important causal aspect of the problems. Those can take as much as 43 iterations to be initially propagated by R_∞. The subsequent propagations are typically faster because they are made incrementally. As expected, a single iteration was needed to converge on all temporally simple problems.

Note that the current usage of our approach in FAPE is limited to restricting the search space. While it proves extremely useful on a wide variety of problems, one could contemplate using the available data structures to extract a heuristic value. The extraction of a relaxed plan has proven to be an effective heuristic in many planners. Earliest possible times $ea^*(a)$ and $ea^*(f)$ could also be used as admissible estimates when considering makespan optimization. However, FAPE's constraint-based algorithm, with POCL and lifted representation, is not yet suitable for this.

6 Related Work

The problem of required concurrency in temporal planning has been analyzed by Cushing et al. (2007). The authors distinguish temporally expressive problems that feature required concurrency from temporally simple problems that do not. Temporally expressive problems are further studied by Cooper, Maris, and Régnier (2013) who identify temporally-cyclic problems in which sets of actions can be interdependent. While the 7th and 8th International Planning Competitions (IPC) included problems with required concurrency, none of those had interdependent actions. In fact, even top performers in the temporal track of the IPC, including Temporal Fast Downward (Eyerich, Mattmüller, and Röger 2012) and YAHSP3 (Vidal 2014), cannot solve problems with interdependent actions.

In classical planners such as FF (Hoffmann and Nebel 2001), the most widely used reachability analysis involves building a Relaxed Planning Graph (RPG) from delete-free actions. CRIKEY3 (Coles et al. 2008), extended this technique to support temporal problems with interdependencies by splitting durative actions into at-start and at-end snap actions. The resulting Temporal RPG is used by CRIKEY3 and its successors POPF (Coles et al. 2010), COLIN (Coles et al. 2012) and OPTIC (Benton, Coles, and Coles 2012) both for reachability analysis and heuristic computation.

Cooper, Maris, and Régnier (2014) discuss another relaxation of temporal planning problems into monotone problems that can be solved in polynomial time. This relaxation is orthogonal to the delete-free relaxation and could also be used for reachability analysis. A key part of this relaxation is the removal of any condition that might be achieved by more than one action; this is likely to lead to poor performance on HTN planning problems where the difficulty is precisely to choose which action to support a given task.

HTN planning systems in the line of SHOP2 (Nau et al. 2003), avoid the need for reachability analysis by (1) manually annotating methods with conditions of applicability; and (2) requiring a total order between all operators, to ensure a method's conditions can be tested on fully defined states. While this technique has proven to be extremely useful on many practical problems, it increases the required domain-engineering effort as well as the risk of introducing modeling errors. Even temporal HTN planners such as SIADEX (Castillo et al. 2006) only partially remove the need for total-order between operators and cannot solve problems with required concurrency.

The recent development of hybrid hierarchical and generative planners such as PANDA (Schattenberg 2009) and FAPE (Dvorak et al. 2014a) has motivated the need for automated search guidance for hierarchical problems. Along these lines, Bercher, Keen, and Biundo (2014) and Elkawkagy et al. (2012) proposed techniques for evaluating the remaining search effort in hierarchical problems by exploiting landmarks and task decomposition graphs. However, hierarchical and causal constraints are still mainly considered independently, resulting in limited heuristic guidance.

Our work shares some conceptual similarities with Angelic Hierarchical Planning (Marthi, Russell, and Wolfe 2007), which performs automated analysis of sequential hierarchical problems in order to infer upper and lower bounds of the set of reachable states. Those sets are used in a hierarchical planner to detect when a task network is always refinable to a solution plan or when it is a dead-end. Those techniques are however only applicable to sequential hierarchical planning. While our technique only focuses on dead-end detection, we consider more general hierarchical problems, featuring concurrency and partial-ordering.

A translation of some hierarchical features of ANML into PDDL was proposed by Smith, Frank, and Cushing (2008) and a complete translation of a restricted class of HTN problems into PDDL was proposed by Alford, Kuter, and Nau (2009). Unlike the exact translations described in those works, we introduce a relaxed translation applicable to any HTN problem for the purpose of reachability analysis. This simpler transformation allows us to avoid the discovery by

Alford et al. (2014) that heuristics based on delete-free relaxation require further relaxation to allow tractability when dealing with hierarchical problems.

7 Conclusion

In this paper, we developed a technique to perform more accurate reachability analysis for temporal planning problems involving interdependent actions. This technique allows us to do a better job of recognizing impossible actions and estimating the earliest start times for actions and fluents. We also showed how interdependencies naturally arise in hierarchical planning problems and introduced a simple relaxation of those problems into temporal planning to enable reachability analysis on hierarchical planning problems.

Our method has been implemented in FAPE, a constraint-based temporal planner for the ANML language. We evaluated the effectiveness of the technique for pruning the search space in both hierarchical and generative planning problems. When compared to state of the art techniques, we showed that our algorithm provides notable improvements on temporally complex problems while having no computational overhead on temporally simple ones. This characteristic makes our method suitable for reachability analysis in a wide range of temporal as well as hierarchical problems.

Acknowledgements. We would like to thank Malik Ghallab and Félix Ingrand for their valuable comments on early versions of this paper. This work was supported in part by the EDSYS Doctoral School of the University of Toulouse, Stinger Ghaffarian Technologies (SGT) Incorporated, the EU MUMMER project funded by the H2020 program under grant agreement No 688147, the NASA Safe Autonomous Systems Operations (SASO) project, and the NASA Autonomous Systems and Operations Project.

References

Alford, R.; Shivashankar, V.; Kuter, U.; and Nau, D. S. 2014. On the Feasibility of Planning Graph Style Heuristics for HTN Planning. In *Proc. of the 24th Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Alford, R.; Kuter, U.; and Nau, D. S. 2009. Translating HTNs to PDDL: A Small Amount of Domain Knowledge Can Go a Long Way. In *Proc. of the 21st Int. Joint Conf. on Artificial Intelligence (IJCAI)*.

Benton, J.; Coles, A.; and Coles, A. 2012. Temporal Planning with Preferences and Time-Dependent Continuous Costs. *Proc. of the 22th Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid Planning Heuristics Based on Task Decomposition Graphs. In *Proc. of the Seventh Annual Symp. on Combinatorial Search (SOCS)*.

Castillo, L. A.; Fernández-Olivares, J.; García-Pérez, Ó.; and Palao, F. 2006. Efficiently Handling Temporal Knowledge in an HTN Planner. In *Proc. of the 16th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 63–72.

Coles, A.; Fox, M.; Long, D.; and Smith, A. 2008. Planning with Problems Requiring Temporal Coordination. *Proc. of the 22th AAAI Conf. on Artificial Intelligence*.

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order Planning. In *Proc. of the 20th Int. Conf. on Automated Planning and Scheduling (ICAPS)*, 42–49.

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2012. COLIN: Planning with Continuous Linear Numeric Change. *Journal of Artificial Intelligence Research (JAIR)* 44:1–96.

Cooper, M.; Maris, F.; and Régnier, P. 2013. Managing Temporal Cycles in Planning Problems Requiring Concurrency. *Computational Intelligence* 29(1):111–128.

Cooper, M.; Maris, F.; and Régnier, P. 2014. Monotone temporal planning: Tractability, extensions and applications. *Journal of Artificial Intelligence Research* 50:447–485.

Cushing, W.; Kambhampati, S.; Weld, D. S.; et al. 2007. When is temporal planning really temporal? In *Proc. of the 20th Int. Joint Conf. on Artifical Intelligence*, 1852–1859.

Dvorak, F.; Barták, R.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014a. Planning and Acting with Temporal and Hierarchical Decomposition Models. In *Proc. of the 26th IEEE Int. Conf. on Tools with Artificial Intelligence, ICTAI*, 115–121.

Dvorak, F.; Bit-Monnot, A.; Ingrand, F.; and Ghallab, M. 2014b. A Flexible ANML Actor and Planner in Robotics. In *Planning and Robotics (PlanRob) Workshop (ICAPS)*.

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving Hierarchical Planning Performance by the Use of Landmarks. In *Proc. of the 26th AAAI Conf. on Artificial Intelligence*.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2012. Using the context-enhanced additive heuristic for temporal and numeric planning. *Springer Tracts in Advanced Robotics*.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. Elsevier.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 253–302.

Marthi, B.; Russell, S.; and Wolfe, J. 2007. Angelic Semantics for High-Level Actions. In *Proc. of the 17th Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Nau, D.; Au, T.-c.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research (JAIR)* 20:379–404.

Schattenberg, B. 2009. *Hybrid planning & scheduling*. Ph.D. Dissertation, University of Ulm.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML language. In *The ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.

Vidal, V. 2014. YAHSP3 and YAHSP3-MT in the 8th International Planning Competition. In *8th International Planning Competition (IPC-2014)*.

Cost-Optimal Algorithms for Hierarchical Goal Network Planning: A Preliminary Report

Vikas Shivashankar[1] vikas.shivashankar@knexusresearch.com
Ron Alford[2] ralford@mitre.org
Mark Roberts[3] mark.roberts.ctr@nrl.navy.mil
David W. Aha[4] david.aha@nrl.navy.mil

[1]Knexus Research Corporation, National Harbor, MD
[2]MITRE, McLean, VA
[3]NRC Postdoctoral Fellow, Naval Research Laboratory, Code 5514, Washington DC
[4]Naval Research Laboratory, Code 5514, Washington DC

Abstract

There is an impressive body of work in developing search heuristics and other reasoning algorithms to guide domain-independent planning algorithms towards (near-) optimal solutions. However, very little effort has been expended in developing analogous techniques to guide search towards high-quality solutions in domain-configurable planning formalisms, such as HTN planning. In lieu of such techniques, the domain-specific knowledge often needs to provide the necessary search guidance to the planning algorithm; this not only imposes a significant burden on the domain author, but can also result in brittle or error-prone domain models.

This work attempts to address this gap by extending recent work on a new hierarchical planning formalism called Hierarchical Goal Network (HGN) Planning to develop the ***H**ierarchically-**Op**timal **G**oal **D**ecomposition **P**lanner (*HOpGDP*)*, a HGN planning algorithm that computes *hierarchically-optimal* plans. HOpGDP is guided by h_{HL}, a new HGN planning heuristic that extends existing admissible landmark-based heuristics from Classical Planning in order to compute admissible cost estimates for HGN planning problems. Preliminary experimental results show that our planner compares favorably to the current state-of-the-art.

1 Motivation and Background

A primary research focus in AI planning is developing efficient search heuristics and auxiliary reasoning techniques that can help the planner find high-quality plans efficiently. Formalisms for automated planning developed in the literature to represent and solve planning problems broadly fall into either *domain-independent planning* or *domain-configurable planning*. Domain-independent planning formalisms, such as *classical planning* requires that the users only provide models of the base actions executable in the domain. In contrast, domain-configurable planning formalisms such as *Hierarchical Task Network (HTN) Planning* allow users to supplement the action models with additional domain-specific knowledge structures that increases the expressivity and scalability of the planning systems.

An impressive body of work exploring search heuristics that has helped scale up search for high-quality solutions in classical planning. Concretely, search heuristics such as the relaxed planning graph heuristic (Hoffmann and Nebel 2001), landmark generation algorithms (Hoffmann, Porteous, and Sebastia 2004; Richter and Westphal 2010), and landmark-based heuristics (Richter and Westphal 2010; Karpas and Domshlak 2009) dramatically improved optimal and anytime planning algorithms by guiding search towards (near-) optimal solutions to planning problems.

Yet, relatively little effort has been devoted to develop analogous techniques to guide search towards high-quality solutions in domain-configurable planning systems. In lieu of such search heuristics, domain-configurable planners often require additional domain-specific knowledge to provide the necessary search guidance. This requirement not only imposes a significant burden on the user, but also sometimes leads to brittle or error-prone domain models. To address this gap, this paper leverages recent work on a new hierarchical planning formalism called *Hierarchical Goal Network* (HGN) Planning (Shivashankar et al. 2012; 2013), which combines the hierarchical structure of HTN planning with the goal-based nature of classical planning.

In this paper, we develop the ***H**ierarchically-**Op**timal **G**oal **D**ecomposition **P**lanner (*HOpGDP*)*, a HGN planning algorithm that uses admissible heuristic estimates to generate *hierarchically-optimal* plans, i.e plans that are both valid and optimal with respect to the given hierarchical knowledge. In particular, our contributions are as follows:

- **Admissible Heuristic**: We present h_{HL}(**H**GN **L**andmark heuristic), a HGN planning heuristic that extends landmark-based admissible heuristics from classical planning to derive admissible cost estimates for HGN planning problems. To the best of our knowledge, h_{HL} is the first admissible heuristic for hierarchical planning[1].
- **Optimal Planning Algorithm**: We describe HOpGDP, an A* search algorithm that uses h_{HL} to generate *hierarchically-optimal* plans.
- **Preliminary Experimental Results**: We provide preliminary experimental evidence showing that HOpGDP outperforms optimal classical planners due to its ability to exploit hierarchical knowledge. We also see that h_{HL} provides useful search guidance by showing that it compares favorably both in terms of runtime and nodes explored to HOpGDP$_{\text{blind}}$, the variant of HOpGDP that uses the trivial heuristic $h = 0$, despite a significant computation overhead.

[1]We are of course not counting the trivial heuristic of $h = 0$.

2 Preliminaries

In this section we detail the classical planning model, review how landmarks are constructed for classical planning and an admissible landmark-based heuristic h_L, and describe goal network planning using examples from assembly planning.

2.1 Classical Planning

We define a *classical planning domain* $D_{classical}$ as a finite-state transition system in which each state s is a finite set of ground atoms of a first-order language L, and each action a is a ground instance of a planning operator o. A planning operator is a 4-tuple $o = (\text{head}(o), \text{precond}(o), \text{effects}(o), \text{cost}(o))$, where precond($o$) and effects($o$) are conjuncts of literals called o's *preconditions* and *effects*, and head(o) includes o's *name* and *argument list* (a list of the variables in precond(o) and effects(o)). cost(o) represents the non-negative cost of applying operator o.

Actions. An action a is executable in a state s if $s \models$ precond(a), in which case the resulting state is $\gamma(a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$, where $\text{effects}^+(a)$ and $\text{effects}^-(a)$ are the atoms and negated atoms, respectively, in effects(a). A plan $\pi = \langle a_1, \ldots, a_n \rangle$ is executable in s if each a_i is executable in the state produced by a_{i-1}; and in this case $\gamma(s, \pi)$ is the state produced by executing the entire plan. If π and π' are plans or actions, then their concatenation is $\pi \circ \pi'$.

We define the *cost* of $\pi = \langle a_1, \ldots, a_n \rangle$ as the sum of the costs of the actions in the plan, i.e. $\text{cost}(\pi) = \sum_{i \in \{1 \ldots n\}} a_i$.

2.2 Generating Landmarks for Classical Planning

There are several landmark generation algorithms suggested in the literature, such as (Hoffmann, Porteous, and Sebastia 2004) and LAMA (Richter and Westphal 2010). The general approach used in generating sound landmarks is to relax the planning problem, generate sound landmarks for the relaxed version, and then use those for the original planning problem. In this paper, we use LAMA's landmark generation algorithm, which uses relaxed planning graphs and domain-transition graphs in tandem to generate landmarks.

2.3 h_L: an Admissible Landmark-based Heuristic

We provide some background on h_L, the landmark-based admissible heuristic for classical planning problems proposed by Karpas and Domshlak (Karpas and Domshlak 2009) that we will be using in our heuristic.

Consider a classical planning problem $P = (D, s_0, g)$ and a landmark graph $LG = (L, Ord)$ computed using any off-the-shelf landmark generation algorithms (e.g., LAMA (Richter and Westphal 2010)). Then, we can define $\mathsf{Unreached}(L, s, \pi) \subseteq L$ to be the set of landmarks that need to be achieved from s onwards, assuming we got to s using the plan π. Note that $\mathsf{Unreached}(L, s, \pi)$ is path-dependent: it can vary for the same state when reached by different paths. It can be computed as follows:

$$\mathsf{Unreached}(L, s, \pi) = L \setminus (\mathsf{Accepted}(L, s, \pi) \setminus \mathsf{ReqAgain}(L, s, \pi))$$

where $\mathsf{Accepted}(L, s, \pi) \subseteq L$ is the set of landmarks that were true at some point along π. $\mathsf{ReqAgain}(L, s, \pi) \subseteq L$ is the set of landmarks that were accepted but are required again; an accepted landmark l is required again if (1) it does not hold true in s, and (2) it is greedy-necessarily ordered before another landmark l' in L that is not accepted.

Karpas and Domshlak show that it is possible to partition the costs of the actions A in D over the landmarks in $\mathsf{Unreached}(L, s, \pi)$ to derive an admissible cost estimate for the state s as follows: let $cost(\phi)$ be the cost assigned to the landmark ϕ, and $cost(a, \phi)$ be the portion of a's cost assigned to ϕ. Furthermore, let us suppose these costs satisfy the following set of inequations:

$$\forall a \in A : \sum_{\phi \in \mathsf{Unreached}(a|L,s,\pi)} cost(a, \phi) \leq \mathsf{cost}(a)$$
$$\forall \phi \in \mathsf{Unreached}(L, s, \pi) : cost(\phi) \leq \min_{a \in ach(\phi|s,\pi)} cost(a, \phi) \quad (1)$$

where $ach(\phi|s, \pi) \subseteq A$ is the set of possible achievers of ϕ along any suffix of π, and $ach(a|L, s, \pi) = \{\phi \in \mathsf{Unreached}(L, s, \pi) | a \in ach(\phi|s, \pi)\}$.

Informally, what these equations are encoding is a scheme to partition the cost of each action across all the landmarks it could possibly achieve, and assigns to each landmark ϕ a cost no more than the minimum cost assigned to ϕ by all its achievers. Given this, they prove the following useful claim:

Lemma 1. *Given a set of action-to-landmark and landmark-to-action costs satisfying Eqn. 1,* $h_L(L, s, \pi) = cost(\mathsf{Unreached}(L, s, \pi)) = \sum_{\phi \in \mathsf{Unreached}(L,s,\pi)} cost(\phi)$ *is an admissible estimate of the optimal plan cost from s.*

Note that the choice of exactly how to do the cost-partitioning is left open. One of the schemes Karpas and Domshlak propose is an *optimal cost-partitioning* scheme that uses an LP solver to solve the constraints in Eqn. 1 with the objective function $\max \sum_{\phi \in L(s,\pi)} cost(\phi)$. This has the useful property that given two sets of landmarks L and L', if $L \subseteq L'$, then $h_L(L, s, \pi) \leq h_L(L', s, \pi)$. In other words, the more landmarks you provide to h_L, the more informed the heuristic estimate.

2.4 Goal Networks and HGN Methods

We extend the definitions of (Shivashankar et al. 2012) of HGN planning to work with partially-ordered sets of goals, which we call a goal network.

A *goal network* is a way to represent the objective of satisfying a partially ordered multiset of goals. Formally, it is a pair $gn = (T, \prec)$ such that:

- T is a finite nonempty set of nodes;
- each node $t \in T$ contains a *goal* g_t that is a DNF (disjunctive normal form) formula over ground literals;
- $\prec$ is a partial order over T.

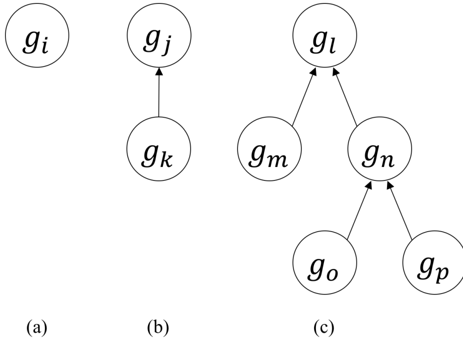


Figure 1: Three generic goal networks we use for examples of the various relationships within a goal network.

We will provide examples of both generic and concrete goal networks. Figure 1 shows three generic goal networks. Each subfigure is itself a goal network denoted gn_a, gn_b, gn_c. Directed arcs indicate a subgoal pair (e.g., (g_k, g_j) from gn_b) such that the first goal must be satisfied before the second goal. Consider the network gn_b where g_k is a subgoal of g_j, then $gn_b = (\{g_j, g_k\}, (g_k \prec g_j))$. Network gn_c shows a partial ordering, where $(\{g_m, g_n\} \prec g_l)$. Similarly, $(\{g_o, g_p\} \prec g_n)$ and this implies both must occur before g_l. Consider a network gn_x that is composed of gn_a and gn_b. Then $gn_x = (\{g_i, g_j, g_k\}, g_k \prec g_j)$. Note that gn_x is a partially ordered forest of goal networks.

Figure 2 shows a concrete goal network for an automated manufacturing domain. $\mathsf{joined}(x, y)$ denotes the goal of assembling the parts x and y together, while $\mathsf{at}(x, loc)$ represents the goal of getting x to location loc. In this goal network, the two goals $\mathsf{joined}(p_2, p_1)$ and $\mathsf{joined}(p_3, p_1)$ are unordered with respect to one another. Furthermore, $\mathsf{joined}(p_2, p_1)$ has three subgoals that need to be achieved before achieving it, i.e the goals of getting the parts p_1, p_2 and the *tool* to the assembly table. These subgoals are also unordered with respect to one another, indicating that the goals can be accomplished in any order.

HGN Methods An *HGN method* m is a 4-tuple (head(m), goal(m), precond(m), network(m)) where the head head(m) and preconditions precond(m) are similar to those of a planning operator. goal(m) is a conjunct of literals representing the goal m decomposes. network(m) is the goal network that m decomposes into. By convention, network(m) has a last node t_g containing the goal goal(m) to ensure that m accomplishes its own goal.

Figure 3 describes the goal network that the $\mathsf{deliver\text{-}obj}$ method decomposes a goal into. This method is relevant to $\mathsf{at}(x, loc)$ goals (since that's the last node), and its preconditions are precond($\mathsf{deliver\text{-}obj}$) = $\{\neg\mathsf{reserved}(agent), \mathsf{can\text{-}carry}(agent, p) \ldots\}$.

Whether a node has predecessors impacts the kinds of operations we allow. We refer to any node in a goal network gn having no predecessors as an *unconstrained* node of gn, otherwise the node is *constrained*. The constrained nodes of Figure 1 include g_j, g_l, g_n and the remaining are unconstrained. The unconstrained nodes in Figure 2 include all the at nodes as well as the $\mathsf{joined}(p3, p1)$ node.

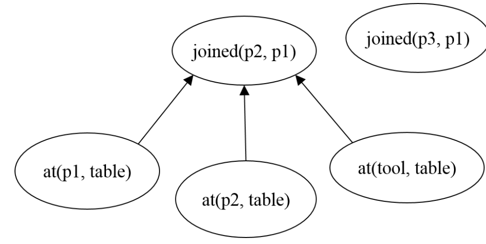


Figure 2: Sample Goal Network for an Automated Manufacturing domain

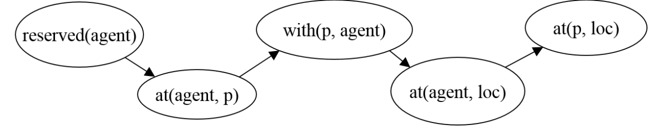


Figure 3: Subgoal network of $\mathsf{deliver\text{-}obj}(p, loc, agent)$, a HGN method to deliver the part p to loc using $agent$.

We define the following operations over any goal network $gn = (T, \prec)$:

1. **Goal Release**: Let $t \in T$ be an unconstrained node. Then the removal of t from gn, denoted by $gn - t$, results in the goal network $gn' = (T', \prec')$ where $T' = T \setminus \{t\}$ and $\prec'$ is the restriction of $\prec$ to T'.
2. **Method Application**: Let $t \in T$ be an unconstrained node. Also, let m be a method applied to t with network(m) $= (T_m, \prec_m)$. Finally, recall that network(m) always contains a 'last' node that contains goal(m); let t_g be this node. Then the application of m to gn via t, denoted by $gn \circ_t m$, results in the goal network $gn' = (T', \prec')$ where $T' = T \cup T_m$ and $\prec' = \prec \cup \prec_m$ $\cup \{(t_g, t)\}$. Informally, this operation adds the elements of network(m) to gn, preserving the order specified by subgoals(m) and setting goal(m) as a predecessor of t.

2.5 HGN Domains, Problems and Solutions

A *HGN domain* is a pair $D = (D_{classical}, M)$ where $D_{classical}$ is a classical planning domain and M is a set of HGN methods.

A *HGN planning problem* is a triple $P = (D, s_0, gn_0)$, where D is a HGN domain, s_0 is the initial state, and $gn_0 = (T, \prec)$ is the initial goal network.

Definition 2 (Solutions to HGN Planning Problems)**.** *The set of* solutions *for P is defined as follows:*

Base Case. *If T is empty, the empty plan is a solution for P.*

In the following cases, let t $\in T$ *be an unconstrained node.*

Unconstrained Goal Satisfaction. *If $s_0 \models g_t$, then any solution for $P' = (D, s_0, gn_0 - t)$ is also a solution for P.*

Action Application. *If action a is applicable in s_0 and a is relevant to g_t, and π is a solution for $P' = (D, \gamma(s_0, a), gn_0)$, then $a \circ \pi$ is a solution for P.*

Method Decomposition. *If m is a method applicable in s and relevant to g_t, then any solution to $P' = (D, s_0, gn_0 \circ_t m)$ is also a solution to P.*

Note that HGN planning allows an action to be applied only if it is *relevant* to an unconstrained node in gn; this prevents action chaining as done in classical planning and allows for tigher control of solutions as in HTN planning. In fact, prior work (Shivashankar et al. 2012) showed that HGN planning is as expressive as HTN planning when both are restricted to totally-ordered methods, i.e. the subtask/subgoal networks are totally ordered.

Let us denote $\mathcal{S}(P)$ as the set of solutions to a HGN planning problem P as allowed by Definition 2. Then we can define what it means for a solution π to be *hierarchically optimal* with respect to P as follows:

Definition 3 (Hierarchically Optimal Solutions)**.** *A solution $\pi^{h,*}$ is hierarchically optimal with respect to P if $\pi^{h,*} = argmin_{\pi \in \mathcal{S}(P)} \mathsf{cost}(\pi)$.*

3 h_{HL}: An Admissible Heuristic for HGN Planning

At a high level, we will proceed to construct h_{HL} as follows:

1. We define a relaxation of HGN planning that ignores the provided methods and allows unrestricted action chaining as in classical planning, which expands the set of allowed solutions,
2. We will extend landmark generation algorithms for classical planning problems to compute sound landmark graphs for the relaxed HGN planning problems, which in turn are sound with respect to the original HGN planning problems as well, and finally
3. We will use admissible classical planning heuristics like h_L on these landmark graphs to compute admissible cost estimates for HGN planning problems.

3.1 Relaxed HGN Planning

Definition 4 (Relaxed HGN Planning)**.** *A relaxed HGN planning problem is a triple $P = (D_{classical}, s_0, gn_0)$ where D is a classical planning domain, s_0 is the initial state, and gn_0 is the initial goal network. Any sequence of actions π that is executable in state s_0 and achieves the goals in gn_0 in an order consistent with the constraints in gn_0 is a valid solution to P.*

Relaxed HGN planning can thus be viewed as an extension of classical planning to solve for goal networks, where there are no HGN methods and the objective is to generate sequences of actions that satisfy the goals in gn_0 in an order consistent with gn_0. In fact, it is easy to show that relaxed HGN planning, in contrast to HGN planning, is no more expressive than classical planning, and relaxed HGN planning problems can be compiled into classical planning problems quite easily.

Next, we will show how to leverage landmark generation algorithms for classical planning to generate landmark graphs for relaxed HGN planning.

3.2 Generating Landmarks for Relaxed HGN Planning

This section describes a landmark discovery technique that can use any landmark discovery technique for classical planning (referred to as LMGEN_C here) such as (Richter and Westphal 2010) to compute landmarks for relaxed HGN planning problems. The main difference here is that while classical planning problems are $(state, goal)$ pairs, relaxed HGN planning problems are $(state, goal\text{-}network)$ pairs; every goal in the goal network can be thought of as a landmark. Therefore, there is now a partially ordered set of goals to compute landmarks from, as opposed to a single goal in classical planning.

Algorithm 1 Procedure for computing landmarks for relaxed HGN planning problems.

```
1: function computeHGNLandmarks(s, gn)
2:     queueSeeds ← gn
3:     queue ← ∅
4:     while queueSeeds is not empty do
5:         choose a g w/o successors from queueSeeds,
       and remove it along with all associated orderings
6:         addLM(g), add g to queue
7:         add any orderings g shares with other goals from
       gn already added to LG
8:         while queue is not empty do
9:             pop landmark ψ from queue and use
       LMGEN_C to generate the new set of landmarks Φ
10:            for φ ∈ Φ do ADDLM(φ, φ →_gn ψ)
11:    return LG
12:
13: function addLM(φ)
14:    if φ is a fact and ∃φ' ∈ LG : φ' ≠ φ ∧ φ ⊨ φ' then
15:        remove φ' from LG and all orderings it is part of
16:    if ∃φ' ∈ LG : φ' ⊨ φ then return φ'
17:    if φ ∉ LG then add φ to queue and return φ
18:
19: function addLMandOrdering(φ, φ →_x ψ)
20:    η ← addLM(φ)
21:    add ordering η →_x ψ to LG
```

We therefore need to generalize classical planning landmark generation techniques to work for relaxed HGN planning problems. The computeHGNLandmarks algorithm (Algorithm 1) describes one such generalization. At a high level, computeHGNLandmarks proceeds by computing landmark graphs for each goal g in gn (which in fact is a classical planning problem) and merging them all together to create the final landmark graph LG.

computeHGNLandmarks takes as input a relaxed HGN planning problem (s, gn) and generates LG, a graph of landmarks. First, queueSeeds is initialized with a copy of gn (Line 2). This is because unlike in classical planning where

we have a single goal to generate landmarks from, in HGN planning we have a partially ordered set of goals to seed the landmark generation; queueSeeds stores these seeds. We also initialize queue, the openlist of landmarks, to $\emptyset$.

While there is a goal g from gn that we have not yet computed landmarks for (Line 4), we do the following: we remove it from queueSeeds along with all induced orderings and add it to queue (Lines 5–6). We also add g to LG using addLM; we also add any ordering constraints it might have with other elements of gn that have already been added to LG. This queue is then used as a starting point by LMGEN_C to begin landmark generation. We iteratively use LMGEN_C to pop landmarks off the queue and generate new landmarks by backchaining until we can no longer generate any more landmarks (Lines 8–10). Each new landmark is added to LG by the addLMandOrdering procedure. Once all goals in gn have been handled, the landmark generation process is completed and the algorithm returns LG.

The addLM procedure takes as input a computed landmark ϕ, adds it to LG and returns a landmark η. There are three cases to consider:

- ϕ subsumes another landmark ϕ' in LG, implying we can remove ϕ' and replace it with ϕ (since ϕ is a stronger version of ϕ'), and return ϕ (Lines 14–15)
- ϕ is subsumed by another landmark ϕ' in LG, implying we can ignore ϕ (Lines 16). In this case, we don't add any new landmark to LG and simply return ϕ'
- ϕ is a new landmark, in which case we can simply add it to LG and return ϕ (Lines 17)

The addLMandOrdering procedure takes as input a landmark ϕ and an ordering constraint $\phi \rightarrow_x \psi$ and adds them to LG. More precisely, it adds ϕ to LG using addLM, which returns the added landmark η. It then adds the ordering constraint between η and ψ in LG.

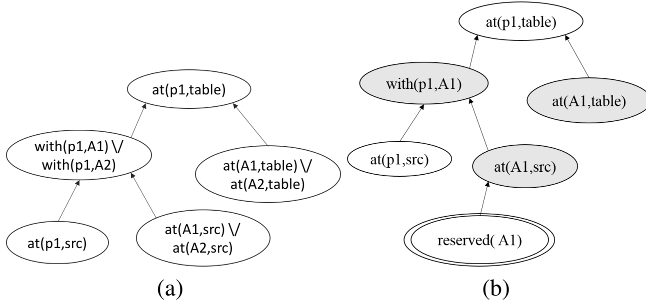


Figure 4: (a) LM graph on goal network containing a single goal $\mathsf{at}(p_1, table)$. (b) LM graph after decomposing $\mathsf{at}(p_1, table)$ with $\mathsf{deliver\text{-}obj}(p_1, table, A1)$. The double-circled landmarks represent new landmarks inferred after the method decomposition, while the landmarks colored gray are new landmarks that subsumed an existing one in (a).

LM graph computation example. Figure 4 illustrates the working of computeHGNLandmarks. Let us assume the goal network gn contains only one goal $g = \mathsf{at}(p_1, table)$. Figure 4a illustrates the output of computeHGNLandmarks on g. This is identical to what LMGEN_C would generate, since gn contains only one goal, making the relaxed HGN problem equivalent to a classical planning problem.

Now, let us assume that we decompose gn using the $m = \mathsf{deliver\text{-}obj}(p_1, table, A1)$, and get the new goal network gn', which essentially looks like an instantiated version of the network in Figure 3. Now if we run computeHGNLandmarks on gn', we end up generating the landmark graph in Figure 4b, which is a more focused version of the first landmark graph. This is because the goals in gn' are landmarks that must be accomplished, which constrains the set of valid solutions that can be generated. For instance, since we've committed to agent $A1$, every solution we can generate from gn' will involve the use of $A1$. We can, as a result, generate more focused landmarks than we otherwise could have from just the top-level goal g. This includes fact landmarks that replace disjunctive landmarks (the ones in gray in Fig. 4b) as well as completely new landmarks that arise as a result of the method; e.g. $\mathsf{reserved}(A1)$ is not a valid landmark for gn, but is one for gn'.

An important point to note at this point is that the subgoals in gn' are not *true* landmarks for g; they are landmarks once we commit to applying method m. However, this actually ends up being useful to us, since it allows us to generate different landmark graphs for different methods; for instance, if we had committed to $A2$, we would have obtained a different set of landmarks specific to $A2$. Now, landmark-based heuristics when applied to these two graphs would get us different heuristic estimates, thus allowing to differentiate between these two methods by using the specific subgoals each method introduces.

It is easy to show that computeHGNLandmarks generates sound landmark graphs for relaxed HGN planning problems:

Claim 5. *Given a relaxed HGN planning problem* $P = (D_{classical}, s_0, gn_0)$, $LG =$ computeHGNLandmarks(s_0, gn_0) *is a sound landmark graph for* P.

Let $P = ((D_{classical}, M), s_0, gn_0)$ be a HGN planning problem, and let $P' = (D_{classical}, s_0, gn_0)$ be the corresponding relaxed version. Then by definition, any solution to P is a solution to P'. Therefore, it is easy to see that a landmark of P' is also a sound landmark of P. More generally, a landmark graph generated for P' is going to be sound with respect to P as well:

Claim 6. *Given a HGN planning problem* P*, then* $LG =$ computeHGNLandmarks(s_0, gn_0) *is a sound landmark graph for* P.

3.3 Computing h_{HL}

The main insight behind h_{HL} is the following: *since the* computeHGNLandmarks *algorithm generates sound landmarks and orderings for relaxed (and therefore regular) HGN planning problems, we can use any admissible landmark-based heuristic from classical planning to derive an admissible cost estimate for HGN planning problems.*

In particular, h_{HL} uses h_L as follows: given an HGN search node (s, gn), the landmark graph is given by

$LG_{HGN} = \text{computeHGNLandmarks}(s, gn)$. Then

$$h_{HL}(s, gn, \pi) = h_L(LG_{HGN}, s, \pi) \quad (2)$$

where π is the plan generated to get to (s, gn).

3.4 Admissibility of h_{HL}

Claim 6 shows that given a HGN problem $P = (D, s_0, gn_0)$, $LG = \text{computeHGNLandmarks}(s_0, gn_0)$ is a sound landmark graph with respect to P. Furthermore, Lemma 1 shows that $h_L(LG, s_0, \langle\rangle)$ provides an admissible cost estimate of the optimal plan starting from s_0 that achieves all the landmarks in LG. Since every solution to P has to achieve all the landmarks in LG in a consistent order, $h_L(LG, s_0, \langle\rangle)$ provides an admissible estimate of the optimal cost to P as well. However, from Eq. 2, $h_L(LG, s_0, \langle\rangle) = h_{HL}(s_0, gn_0, \langle\rangle)$. Therefore, we have the following theorem:

Theorem 7 (Admissibility of h_{HL})**.** *Given a HGN planning domain D, a search node (s, gn, π) and its cost-optimal solution $\pi^{*,HGN}_{s,gn}$, $h_{HL}(s, gn, \pi) \leq \pi^{*,HGN}_{s,gn}$.*

4 The HOpGDP Algorithm

Algorithm 2 describes HOpGDP. It takes as input a HGN domain $D = (D', M)$, the initial state s_0 and the initial goal network gn_0. It returns a plan if it finds one, or failure if the problem is unsolvable.

Initialization. It starts off by initializing open (Line 2), which is a priority queue that sorts the HGN search nodes yet to be expanded by their f-value, where $f((s, gn, \pi)) = \text{cost}(\pi) + h_{HL}(s, gn)$. open initially contains the initial search node $(s_0, gn_0, \langle\rangle)$. It also initializes searchSpace (Line 3), the set of all nodes seen during the search process. This data structure keeps track of the best known path currently known for each state,goal-network pair, and is thus helpful to detect when we find a cheaper path to a previously seen HGN search node.

Search. HOpGDP now proceeds to do an A* search in the space of HGN search nodes starting from the initial node. While open is not empty, it does the following (Lines 4–14): it removes the HGN search node $N = (s, gn, \pi)$ with the best f-value from open (Line 5) and first checks if gn is empty (Line 6). If this is true, this means that all the goals in gn_0 have been solved, and π is the optimal solution to the HGN planning problem.

If gn is not empty, then the algorithm proceeds by using the getSuccessors subroutine to compute N's successor nodes (Line 7). For each successor node (s', gn', π'), it proceeds to do the following: it checks to see if another path η to (s', gn') exists in searchSpace (Line 9). If this is the case and if η is costlier than π' (Line 10), it updates searchSpace with the new path; and reopens the search node (Line 14); if η is cheaper than the new plan π', it simply skips this successor (Line 12).

If (s', gn') has not been seen before, it adds $N' = (s', gn', \pi')$ to searchSpace to track the currently best-known plan π' to (s', gn') (Line 13). It also evaluates the f-value of N' and adds it to open (Line 14).

If there are no more nodes left in open, this implies that it has exhausted the search space without finding a solution, and therefore returns failure (Line 15).

Computing Successors. The procedure getSuccessors computes the successors of a given HGN search node (s, gn, π) in accordance with Definition 2. First, we check to see if there are any unconstrained goals g in gn that are satisfied in the current state s. We then proceed to create new HGN search nodes by removing all such goals from gn (Line 19–20). Next, we compute all actions applicable in s and relevant to an unconstrained goal in gn (Line 21) and create new search nodes by progressing s using these actions (Line 22–23). We compute all pairs (m, g) such that m is a HGN method applicable in s and relevant to an unconstrained goal g in gn (Line 24) and create new search nodes by decomposing g in gn using m (Line 25–26). Finally, we return the set of generated successor nodes (Line 27).

Algorithm 2 Pseudocode of HOpGDP. It takes as arguments the domain description $D = (D_{classical}, M)$, the initial state s_0, and the initial goal network gn_0. It either returns a plan if it finds one, or failure if it doesn't.

1: **function** HOpGDP(D, s_0, gn_0)
2: open $\leftarrow (s_0, gn_0, \langle\rangle)$
3: searchSpace $\leftarrow (s_0, gn_0, \langle\rangle)$
4: **while** open is not empty **do**
5: rem. (s, gn, π) with lowest f-value from open
6: **if** gn is empty **then return** π
7: successors $\leftarrow$ getSuccessors(D, s, gn, π)
8: **for** $(s', gn', \pi') \in$ successors **do**
9: **if** $\exists(s', gn', \eta) \in$ searchSpace **then**
10: **if** cost(π') $<$ cost(η) **then**
11: replace (s', gn', η) with (s', gn', π') in searchSpace
12: **else continue**
13: **else** add (s', gn', π') to searchSpace
14: eval. f-value of (s', gn', π') and add to open
15: **return** failure
16:
17: **function** getSuccessors(D, s, gn, π)
18: successors $\leftarrow \emptyset$
19: **for** unconstrained $g \in gn$ satisfied in s **do**
20: add the node $(s, gn - \{g\}, \pi)$ to successors
21: $\mathcal{A} \leftarrow$ actions in D applicable in s and relevant to an unconstrained goal in gn
22: **for** $a \in \mathcal{A}$ **do**
23: add the node $(\gamma(s, a), gn, \pi \circ a)$ to successors
24: $\mathcal{M} \leftarrow \{(m, g)$ s.t. $m \in M$ is applicable in s and relevant to an unconstrained goal g in $gn\}$
25: **for** $(m, g) \in \mathcal{M}$ **do**
26: add the node $(s, gn \circ_g m, \pi)$ to successors
27: **return** successors

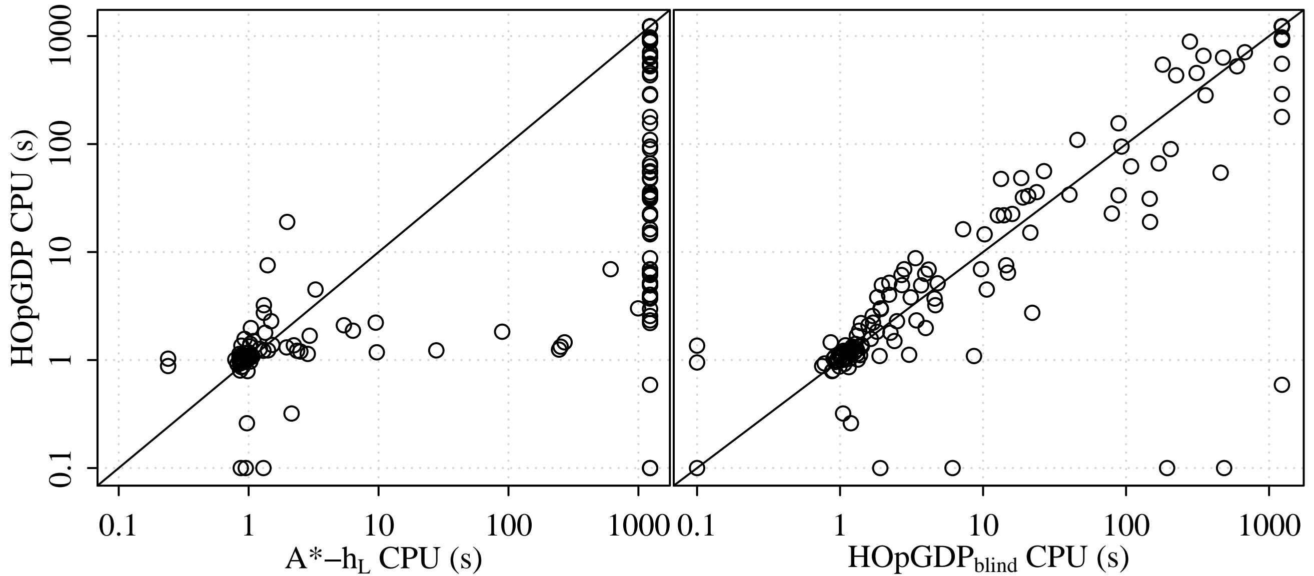


Figure 5: Log-scale scatter plot comparing HOpGDP planning time vs. A*-h_L (left) and HOpGDP$_{\text{blind}}$ (right) planning time on Blocksworld and Logistics problems.

	Count	A*-h_L	HOpGDP$_{\text{blind}}$	HOpGDP
bw	114	60	102	110
log	28	11	22	22
Total	142	71	124	132

	A*-h_L		HOpGDP$_{\text{blind}}$		HOpGDP	
	$\bar{s}$	σ	$\bar{s}$	σ	$\bar{s}$	σ
bw	42.7	157.2	8.0	30.9	1.8	2.6
log	3.1	2.8	1.2	0.2	1.4	0.5

	A*-h_L		HOpGDP$_{\text{blind}}$		HOpGDP	
	$\bar{x}$	σ	$\bar{x}$	σ	$\bar{x}$	σ
bw	1097840	3805838	22047	96523	1194	3806
log	106281	133890	769	775	569	608

Table 1: The coverage (top) and the mean CPU seconds (middle) and mean number of node expansions (bottom) for A*-$h_L$using h_L, HOpGDP$_{\text{blind}}$, and HOpGDPusing h_{HL}. For runtime and nodes, we include the sample mean ($\bar{x}$) and standard deviation (σ). Other than coverage, all statistics are over the subset of problems solved by all three variants.

5 Preliminary Experiments

Our evaluation of HOpGDP focuses on two questions: Is the heuristic informative in guiding search, and is its guidance sufficient to overcome its computation time. We chose the well-known Blocksworld and Logistics domains for our preliminary study, using the HGN methods described in the GoDeL evaluation. For logistics, we limited truck capacity to one package to ensure that the optimal solutions were the same between the HGN and non-HGN planners.

We implemented HOpGDP in the GoDeL framework (Shivashankar et al. 2013), which is derived from the Fast-Downward (Helmert 2009) code base. We chose three variants of HOpGDP to compare: A*-h_L, which ignores all methods and corresponds to A* with the classical h_L heuristic; HOpGDP$_{\text{blind}}$, which corresponds to Algorithm 2 with $h = 0$, so that the f-value is always g, the distance from the start; and the full HOpGDP algorithm with the h_{HL} heuristic. Both A*-h_L and HOpGDP break ties on lower h values. We ran all problems on a Xeon E5-2639 with a per problem limit of 4 GB of RAM and 20 minutes of planning time.

Table 1 (top) shows the coverage for the three algorithms. HOpGDP and HOpGDP$_{\text{blind}}$ have nearly twice the coverage of A*-h_L, which confirms the power of pruning in search, even when comparing blind and heuristic search. The difference between HOpGDP and HOpGDP$_{\text{blind}}$ is more subtle, with HOpGDP covering all of HOpGDP$_{\text{blind}}$'s problems plus eight more.

Figure 5 gives a scatter plot of run times of HOpGDP vs. A*-h_L and HOpGDP$_{\text{blind}}$. Table 1 (middle) summarizes the runtimes for the set of problems solved by all three variants. The results roughly match the coverage trends, but we make

no statistical claims.

Note that the runtime for HOpGDP on logistics is higher than $\mathsf{HOpGDP}_{\mathsf{blind}}$. The bottom section of Table 1 gives some context to the results. A*-h_L has an order of magnitude faster node-expansion rate than $\mathsf{HOpGDP}_{\mathsf{blind}}$, which in turn has a four times higher expansion rate than HOpGDP. Part of the reason for this is architecture. HGN methods, as with HTN methods, tend to have high numbers of free variables. Grounding these methods bloated the domain beyond use in sample runs, and so methods are unified against a state by calling out to SHOP2's unifier over local sockets. Part of the difference between A*-h_L and HOpGDP is inherent, though, since h_L calculates one landmark graph per problem and h_{HL} must calculate a new landmark graph for every goal network it encounters.

While these trends are generally positive, the results are hardly conclusive. We plan on expanding the set of problems and more closely analyze the results in future work.

6 Related Work

HTN planners solve planning problems in one of two ways, either (1) by forward state-space search, such as in the SHOP (Nau et al. 1999) and SHOP2 (Nau et al. 2003) HTN planners, or (2) by partial-order causal-link planning (POCL) techniques, such as in UMCP (Erol, Hendler, and Nau 1994) and in the hybrid planning literature (Elkawkagy et al. 2012; Bercher, Keen, and Biundo 2014).

Due to very little work in the way of search heuristics for forward-search HTN planning, planners often end up providing other domain-specific mechanisms for users to encode search strategies. For example, SHOP2 allows the domain-specific knowledge, known as *HTN methods*, to be specified in a 'good' order according to the user, and tries them out in the same order. SHOP2 also provides support for external function calls (Nau et al. 2003) that can call arbitrary code to do the heavy lifting in the problem, thus minimizing the choices that need to be made during search. For example, in the 2002 Planning Competition for hand-tailored planners, the authors of SHOP2 implemented a shortest-path algorithm in the DriverLog domain that SHOP2 could call externally to generate optimal paths.[2]

Waisbrot et al (Waisbrot, Kuter, and Konik 2008) developed $H2O$, a HTN planner that augments SHOP2 with classical planning heuristics to make local decisions on which method to apply next by estimating how close the method's goal is to the current state. $H2O$, however, retains the depth-first search structure of SHOP2, making it to difficult to generate high-quality plans.

Marthi et al (Marthi, Russell, and Wolfe 2007; 2008) propose an HTN-like formalism called *angelic hierarchical planning* which allows users to annotate abstract tasks with additional domain-specific information in the form of lower and upper bounds on the costs of the possible plans they decompose to. They then use this information to compute hierarchically-optimal plans. In contrast, we require only costs of the primitive actions and use domain-independent search heuristics to compute hierarchically-optimal plans.

[2]personal communication with Ugur Kuter.

There has been recent work on developing search heuristics for POCL HTN planners (Elkawkagy et al. 2012; Bercher, Keen, and Biundo 2014). However, these heuristics typically provide estimates on how many more plan refinement steps need to be taken from a search node in order to get to a solution, as opposed to plan quality estimates, which is what we are focused on in this paper.

Hierarchical Goal Network (HGN) Planning combines the hierarchical structure of HTN planning with the goal-based nature of classical planning. It therefore allows for easier infusion of techniques from classical planning into hierarchical planning, such as adapting the FF heuristic to do *method ordering* in the GDP planner (Shivashankar et al. 2012), and using landmark-based techniques to plan with partial amounts of domain knowledge in GoDeL (Shivashankar et al. 2013). Both planners, however, use depth-first search and inadmissible heuristics, so they cannot provide any guarantees of plan quality.

Another domain-configurable planning formalism is *Planning with Control Rules* (Bacchus and Kabanza 2000), where domain-specific knowledge is encoded in the form of *linear-temporal logic* (LTL) formulas. TLPlan, one of the earliest planners developed under this formalism, used control rules written in LTL to prune away trajectories deemed suboptimal by the user. There have also been attempts to develop heuristic search planners that can plan with LTL_f, a simplified version of LTL that works with finite traces. This has been used to incorporate search heuristics to solve for temporally extended goals written in LTL_f (Baier and McIlraith 2006) as well as to express landmark-based heuristics that guide classical planners (Simon and Roger 2015).

7 Conclusion

Despite the popularity of hierarchical planning techniques both in theory and practice, very little effort has been devoted to developing domain-independent search heuristics that can provide useful search guidance towards high-quality solutions. As a result, end-users need to encode domain-specific heuristics into the domain models, which can make the domain-modeling process tedious and error-prone.

To address this issue, this paper leverages recent work on HGN planning, which allows tighter integration of hierarchical planning and classical planning, to develop (1) h_{HL}, an admissible HGN planning heuristic, and (2) HOpGDP, an A* search algorithm guided by h_{HL} to compute *hierarchically-optimal* plans.

There are several avenues for future work, such as:

- **Theoretical Analysis**: We show that h_{HL} returns admissible cost estimates for HGN planning problems. However, we believe that it has other interesting theoretical properties as well. In particular, we conjecture that it dominates h_L, since it can, in general, compute more focused landmarks, which can translate to more informed heuristic estimates, a property of optimal cost partitioning with h_L. On a related note, we believe that h_{HL} has the nice property that despite some of the steps being zero-cost (i.e. method applications, which don't change the state), it can help us avoid f-value plateaus since the method ap-

plication can result in a more informative heuristic value. We plan on doing a more detailed theoretical analysis to verify these conjectures.

- **Extension to Anytime Planning**: While we believe it is theoretically interesting that h_{HL} can help us find optimal solutions, it would be of practical interest to design *anytime* HGN planning algorithms.

Acknowledgment This work is sponsored in part by OSD ASD (R&E). The information in this paper does not necessarily reflect the position or policy of the sponsors, and no official endorsement should be inferred. Ron Alford performed part of this work under an ASEE postdoctoral fellowship at NRL.

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artif. Intell.* 116:123–191.

Baier, J. A., and McIlraith, S. A. 2006. Planning with first-order temporally extended goals using heuristic search. In *AAAI Conference on Artificial Intelligence*.

Bercher, P.; Keen, S.; and Biundo, S. 2014. Hybrid planning heuristics based on task decomposition graphs. In *Proc. of the Seventh Annual Symposium on Combinatorial Search (SoCS)*, 35–43. AAAI Press.

Elkawkagy, M.; Bercher, P.; Schattenberg, B.; and Biundo, S. 2012. Improving hierarchical planning performance by the use of landmarks. In *AAAI Conference on Artificial Intelligence*, 1763–1769.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. UMCP: A sound and complete procedure for hierarchical task-network planning. 249–254. ICAPS 2009 influential paper honorable mention.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5):503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research* 22:215–278.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In Boutilier, C., ed., *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 1728–1733.

Marthi, B.; Russell, S.; and Wolfe, J. 2007. Angelic semantics for high-level actions. In *International Conference on Automated Planning and Scheduling*.

Marthi, B.; Russell, S.; and Wolfe, J. 2008. Angelic hierarchical planning: Optimal and online algorithms. In *International Conference on Automated Planning and Scheduling*, 222–231.

Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In Dean, T., ed., *International Joint Conference on Artificial Intelligence*, 968–973.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res. (JAIR)* 39:127–177.

Shivashankar, V.; Kuter, U.; Nau, D.; and Alford, R. 2012. A hierarchical goal-based formalism and algorithm for single-agent planning. In *Proc. of the 11th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, volume 2, 981–988. Int. Foundation for Autonomous Agents and Multiagent Systems.

Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. 2013. The GoDeL planning system: a more perfect union of domain-independent and hierarchical planning. In *Proc. of the 23rd Int. Joint Conf. on Artificial Intelligence (IJCAI)*, 2380–2386. AAAI Press.

Simon, S., and Roger, G. 2015. Finding and exploiting ltl trajectory constraints in heuristic search. In *Symposium on Combinatorial Search*.

Waisbrot, N.; Kuter, U.; and Konik, T. 2008. Combining heuristic search with hierarchical task-network planning: A preliminary report. In *International Conference of the Florida Artificial Intelligence Research Society*.

Monte Carlo Tree Search as a Hyper-heuristic Framework for Classical Planning

Otakar Trunda
Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské nám. 25, 118 00 Praha 1, Czech Republic

Abstract

Hyper-heuristics have become popular for solving combinatorial optimization problems, especially in the field of scheduling. The idea of hyper-heuristics is based on an automatic composition of the search algorithm from given building blocks, such that the search algorithm is tailor-made for each problem instance. In planning, this idea has not yet been seriously investigated and it might represent a viable alternative to planning portfolios.

We present a hyper-heuristic planner that can automatically adapt the search strategy to given problem instance. The planner works with a set of simple search algorithms used as building blocks and combines them during the search in order to adapt to the problem instance. We use Monte Carlo Tree Search to manage the combination process of building blocks.

Preliminary experiments show that the price/performance ratio of our technique is advantageous as the proposed hyper-heuristic outperforms each single building-block-algorithm when used individualy.

Introduction

Planning deals with problems of selection and causally ordering of actions to achieve a given goal from a known initial situation. Planning algorithms assume a description of possible actions and attributes of the world states in some modeling language such as Planning Domain Description Language (PDDL) as its input. Currently, the most efficient approach to solve planning problems is heuristic forward search.

Many planning algorithms have been developed so far, none of them outperforms each other on all domains. It is therefore important to select the right algorithm for the task at hand. A similar problem occurs in other fields like combinatorial search and machine learning where it is addressed by so called *autonomous methods*. Among those methods, *hyper-heuristics* are especially preferred for combinatorial optimization problems. In planning, autonomous methods in their true form have not yet been used and the problem of planning algorithm selection is currently only addressed by portfolio planners. Since hyper-heuristics proved to be efficient in fields closely related to planning (like scheduling and combinatorial search (Burke et al. 2013)), it is worth trying to use them for planning as well.

Instead of searching the *solution space* directly, hyper-heuristics search the *algorithm space*. They try to build an algorithm well suited for the problem instance. To evaluate the candidate algorithms, they use the quality of solution that the algorithm finds. There is always a metaheuristic that handles the search for algorithms. From the planning perspective, the Monte Carlo Tree Search algorithm (MCTS) has a distinct advantage over other metaheuristics for being able to implicitly work with sequences of variable length.

Monte Carlo Tree Search algorithm is a stochastic method originally proposed for computer games. MCTS was modified for a single-player games and it is also applicable to optimization problems. However, there are still difficulties when applying to planning problems, namely existence of infinite sequences of actions and dead ends. In this paper, we identify these difficulties and we discuss possible ways to overcome them with a connection to the hyper-heuristic design.

The paper is organized as follows. We will first give a short background on planning, hyper-heuristics and Monte Carlo Tree Search techniques and we will highlight possible problems when applying MCTS in planning including a discussion how to resolve these problems. We will then present the design of our hyper-heuristic planner. The paper will be concluded by preliminary experiments.

Background

Planning

In this paper, we deal with classical planning problems, that is, with finding a sequence of actions transferring the world from a given initial state to a state satisfying a certain goal condition (Ghallab, Nau, and Traverso 2004). *World states* are represented as sets of predicates that are true in the state (all other predicates are false in the state). *Actions* describe how the world state can be changed. Each action a is defined by a set of predicates $prec(a)$ as its precondition and two disjoint sets of predicates $eff^+(a)$ and $eff^-(a)$ as its positive and negative effects. Action a is applicable to state s if $prec(a) \subseteq s$ holds. If action a is applicable to state s then a new state $\gamma(a, s)$ defines the state after application of a as

$$\gamma(a, s) = (s \cup eff^+(a)) \setminus eff^-(a)$$

Otherwise, the state $\gamma(a, s)$ is undefined. The goal g is usually defined as a set of predicates that must be true in the goal state. Hence the state s is a *goal state* if and only if $g \subseteq s$.

The *satisficing planning task* is formulated as follows: given a description of the initial state s_0, a set A of available actions, and a goal condition g, is there a sequence of actions $(a_1, \ldots, a_n)$, called a *solution plan*, such that $a_i \in A$, a_1 is applicable to state s_0, each a_i s.t. $i > 1$ is applicable to state $\gamma(a_{i-1}, \ldots \gamma(a_1, s_0))$, and $g \subseteq \gamma(a_n, \gamma(a_{n-1}, \ldots \gamma(a_1, s_0)))$?

Assume that each action a has some cost $c(a)$. An *optimization planning task* is about finding a solution plan such that the sum of costs of actions in the plan is minimized. Formally, the task is to find a sequence of actions $(a_1, \ldots, a_n)$, minimizing $\sum_{i=1}^{n} c(a_i)$ under the condition
$g \subseteq \gamma(a_n, \gamma(a_{n-1}, \ldots \gamma(a_1, s_0)))$.

Our planning system uses a *finite-domain representation* of the planning task encoded in the SAS^+ formalism instead of the predicate representation described here. For the purposes of this paper, it is sufficient to define the planning problem simply as a state-transition system with transition costs so the internal representation is not that important at this point.

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a stochastic optimization algorithm that combines classical tree search with random sampling of the search space. The algorithm was originally used in the field of game playing where it became very popular, especially for games Go and Hex. A single player variant has been developed by Schadd et al. (Schadd et al. 2008) which is designed specifically for single-player games and can also be applied to optimization problems. The MCTS algorithm successively builds an asymmetric tree to represent the search space by repeatedly performing the following four steps:

1. *Selection* – The tree built so far is traversed from the root to a leaf using some criterion (called *tree policy*) to select the most urgent leaf.
2. *Expansion* – All applicable actions for the selected leaf node are applied and the resulting states are added to the tree as successors of the selected node (sometimes different strategies are used).
3. *Simulation* – a pseudo-random simulation is run from the selected node until some final state is reached (a state that has no successors). During the simulation, the actions are selected by a *simulation policy*,
4. *Update/Back-propagation* – The result of the simulation is propagated back in the tree from the selected node to the root and statistics of the nodes on this path are updated according to the result.

The core schema of MCTS is shown at Figure 1 from (Chaslot et al. 2008).

One of the most important parts of the algorithm is the *node selection criterion* (a tree policy). It determines which node will be expanded and therefore it affects the shape of the search tree. The purpose of the tree policy is to solve the exploration vs. exploitation dilemma.

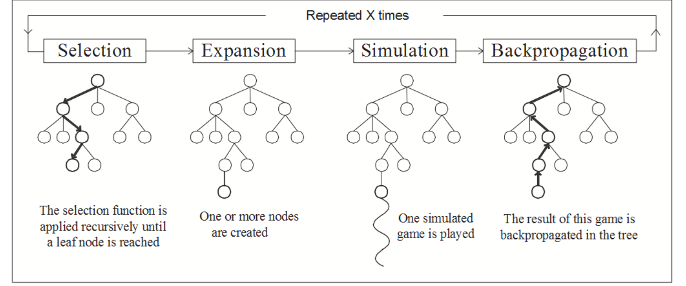


Figure 1: Basic schema of MCTS (Chaslot et al. 2008)

Commonly used policies are based on a so called *bandit problem* and *Upper Confidence Bounds for Trees* (UCT) (Auer, Cesa-Bianchi, and Fischer 2002; Kocsis and Szepesvári 2006) which provide a theoretical background to measure quality of policies. We will present here the tree policy for the single-player variant of MCTS (SP-MCTS) due to Schadd et al. (Schadd et al. 2008) that is appropriate for planning problems (planning can be seen as a single-player game where moves correspond to action selection).

Let $t(N)$ be the number of simulations/samples passing the node N, $v_i(N)$ be the value of i-th simulation passing the node N, and $\bar{v}(N)$ be the average value of all simulations passing the node N:

$$\bar{v}(N) = \frac{\sum_{i=1}^{t(N)} v_i(N)}{t(N)}$$

The SP-MCTS tree policy suggests to select the children node N_j of node N maximizing the following function (called *urgency)*:

$$\bar{v}(N_j) + C \cdot \sqrt{\frac{2 \ln (t(N))}{t(N_j)}} + \sqrt{\frac{\sum_{i=1}^{t(N_j)} v_i(N_j)^2 - t(N_j) \cdot \bar{v}(N_j)^2}{t(N_j)}}$$

The first component of the above formula is a so called *Expectation* and it describes an expected value of the path going through a given node. This supports exploitation of accumulated knowledge about quality of paths. The second component is a *Bias*. The Bias component of a node N_j slowly increases every time the sibling of N_j is selected (that is every time the node enters the competition for being selected but it is defeated by another node) and rapidly decreases every time the node N_j is selected, that is the policy prefers nodes that have not been selected for a long time. This supports exploration of unknown parts of the search tree. *Bias* is weighted by a constant C that determines the *exploration vs. exploitation* ratio. Its value depends on the domain and on other modifications to the algorithm. For example in computer Go the usual value is about 0.2. When

solving optimization problems, the range of values for the *Expectation* component is unknown opposite to computer games, where *Expectation* is between 0 (loss) and 1 (win). Nevertheless it is possible to use an adaptive technique for adjusting the parameter C in order to keep the components in the formula (*Expectation* and *Bias*) of the same magnitude (Baudiš 2011). The last component of the evaluation formula is a standard deviation and it was added by Schadd et al. (Schadd et al. 2008) to improve efficiency for single-player games (puzzles).

Another important part of the algorithm is the *simulation phase*. Simulations are used to sample the search space and their accumulated results play the role of an evaluation function for non-goal states. To work efficiently, the algorithm needs a large number of simulations to be performed every second so it is crucial that simulations are fast and easy to carry out.

The Figure 2 shows an example of the MCTS tree when minimizing a pseudo-random one-dimensional function. In the yellow field, there is the fitness landscape of a function to be minimized and above it there is a tree that MCTS builds. The algorithm correctly identifies promising areas of the search space and focuses the search on those areas.

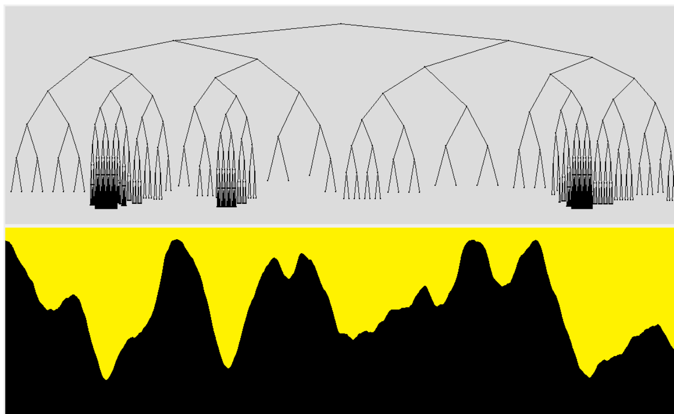


Figure 2: Simple example of a MCTS tree

Autonomous search

The field of autonomous search studies techniques that can automatically decide how to approach a given problem, i.e. what search algorithm to use and how to configure it (Hamadi, Monfroy, and Saubion 2012). Autonomous search unites areas like algorithm selection, hyper-heuristics, parameter tuning and parameter control. They can be divided into two categories: *configuration* and *control*. Configuration techniques determine the proper algorithm and its parameters before the search starts and then uses such algorithm during the search. Control techniques, on the other hand, continuously monitor the ongoing results and adjust the search strategy accordingly.

The need for algorithm selection techniques is apparent in many fields as the efficiency of algorithms greatly depends on the type of problem they are solving. It is clear that there is no ”one correct” algorithm for all problems, but instead different problems require different search techniques. The same holds in the field of planning as demonstrated by recent successes of portfolio-based planners (Borrajo and Linares López 2012).

Algorithm selection techniques work with a set of algorithms and try to select the most suited algorithm for a given task. They often use machine learning approaches trying to classify tasks into categories based on their similarity and then assigning each category the most suitable algorithm. This approach is especially popular for search-algorithm selection (Kumar, Singh, and Kumar 2015) and machine-learning algorithm selection (Kazík et al. 2011).

Automated parameter tuning techniques aim on find a combination of parameters of an algorithm that leads to best performance. They are often used to tune parameters of metaheuristics and machine-learning algorithms (Thornton et al. 2013) where they can significantly increase the performance.

Hyper-heuristics try to build an algorithm suited for given task by combining so called *low-level algorithms* during the search. A pool of simple algorithms is given and the hyper-heuristic combines them into a more complex unit. The combination procedure is often based on some kind of evolutionary computation and the quality of resulting units is measured by how well can they solve the original task. The approach was especially successful in the area of scheduling and it is now applied to many different kinds of problems (Burke et al. 2013). Our method falls into this category as it automatically combines low-level planners into a more complex unit tailor-made for the input problem.

Instead of searching the *solution space* directly, hyper-heuristics search the *algorithm space*. The approach is based on an assumption that *similar algorithms will find solutions of similar quality* which implies that the algorithm space has some favourable properties: high locality (elements close to each other have similar evaluation) and low number of local extrema. On spaces with those properties, optimization metaheuristics can find good solutions quickly. Of course, those improvements come for a price: evaluating an element from the algorithm space takes much more time because it involves searching for solution in the solution space.

In the field of planning, the idea of autonomous search is most eminent in portfolio-planners. There is also an increasing trend of categorizing planning tasks based on a set of features and using such classification for algorithm selection (Fawcett et al. 2014; Vallati, Chrpa, and Kitchin 2014; Cenamor, de la Rosa, and Fernandez 2013). To our best knowledge, no hyper-heuristic approach has been used in the field of classical planning so far.

MCTS for Planning

MCTS is a robust optimization metaheuristic suitable for solving various optimization problems. It is implicitly capable of working with *sequences* which makes it more suitable for optimization planning than other metaheuristics like evolutionary algorithms for example. Most metaheuristics

handle candidate solutions as points in some d-dimensional space. This way, we can easily represent a set of fixed-length sequences, but variable-length sequences might prove difficult. Moreover designing search operators (like cross-over or mutation) that transform valid plans into different but still valid plans is difficult. For example, replacing some action in a plan will most likely make it invalid since the latter actions might become inapplicable. MCTS on the other hand represents the search-space as a tree and traverses it in a forward manner, like most forward planners do.

The planning task can be seen as the problem of finding a shortest path in an implicitly given state space, where transitions/moves between the states are defined by the actions. From this point of view, planning is very close to single-player games though there are some notable differences.

Cycles in the state-space

MCTS uses simulations to evaluate the states. Hence, from the planning perspective, we need to generate solution plans – valid sequences of actions leading to a goal state. Unlike *Hex* and other game applications of MCTS, planning problems allow infinite paths in the state-space (even though the state-space is finite) and this is quite usual in practice since the planning actions are typically reversible.

This is a serious problem for the MCTS algorithm since it causes the expected length of the simulations to be very large (exponential in the distance to the nearest goal state) and therefore only very few simulations can be carried out within a given time limit.

Dead ends and dead components

The other problem is existence of plans that do not lead to a goal state. We use the term *dead end* to denote a state such that no action is applicable to this state and the state is not a goal state. Note that dead ends do not occur in any game domain since in games any state that does not have successors is considered a goal state and has a corresponding evaluation assigned to it (like Win, Loss, or Draw in case of Chess or Hex, or a numerical value in case of SameGame for example). In planning, however, the evaluation function is defined only for the solution plans leading to goal states. A plan that cannot be extended doesn't necessarily have to be a solution plan and it is not clear how to evaluate the simulation that reached a dead end.

A *dead component* is a combination of both previous problems – it is a strongly connected component in the state-space that does not contain any goal state. This is similar to the dead ends problem except that we can easily detect a dead end (since there are no applicable actions there) but it is much more difficult to detect a dead component.

Possible solutions

There are several ways to deal with these problems.

1. *Modifying the state-space* so that it does not contain infinite paths nor dead ends and dead components.
2. *Using a simulation policy* which would guarantee that the simulations reach a goal state quickly, avoiding dead ends.
3. *Setting an upper bound* on the length of simulations.
4. Finding a way to *evaluate the dead end states* and other non-goal states.

Number (1) would be the most efficient way, it is however difficult to find such modifications in general as it involves solving the underlying satisficing planning problem. For some types of domains, such modification is possible using meta-actions as presented in (Trunda 2013; Trunda and Barták 2013).

In this paper, we choose a different way. We develop means to lead the simulations to goal states. As such techniques can't be efficient in all situations, we combine them with setting a limit on the length of the simulations and develop a way to evaluate simulations that reached the limit or ended in a dead end.

Designing a hyper-heuristic based planner

We address the issues with simulations by using a standard planning algorithm in the simulation phase of MCTS. The algorithm should lead the simulations towards goal states and it should do so very fast. Quality of solutions is not an issue here. The simulations should be random samples, they should end in *some* random goal state reachable from the selected MCTS tree node. The search algorithm used as the simulation policy should have following properties:

- lead the simulation from given initial state towards some goal state, avoid dead ends and cycles
- be very fast
- may find (even vastly) suboptimal solutions
- be able to solve any type of planning problem quickly

There is a question of which planning algorithm to use in the simulation phase. The choice is not clear since there are many options and specific algorithms are usually only suited for specific types of domains. We decided to approach the situation as an opportunity to modify MCTS into a hyper-heuristic algorithm. Instead of just one, we use a portfolio of different search algorithms and let MCTS to choose from them on its own. We use the standard *MCTS selection process* for selecting not only promising parts of the search space but also promising search algorithms. Our design looks as follows:

We will use MCTS in a standard way for planning as in (Trunda 2013; Trunda and Barták 2013), that is:

- tree covers an initial part of the problem state-space
- root of the tree represents the initial state
- edges from the node correspond to applicable actions
- successors correspond to states after applying the action
- each node represents a sequence of actions given by labels of edges on the path from root to the node

In the figure 3, there is an example of MCTS tree early during the search. s_0 is the initial state, s_1 is the selected leaf, a_1 to a_3 are actions. In the figure 4, there is the tree after expansion. New states that are reachable from s_1 are added.

The algorithm works as described in previous section. It selects the most urgent leaf, expands it by adding its successors to the tree and then runs a simulation from this leaf. Simulations corresponds to finding some path from a state that the leaf represents to some goal state.

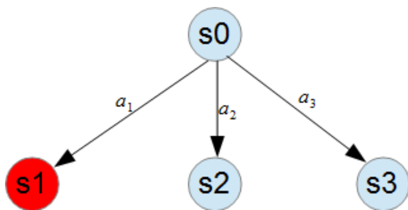


Figure 3: Example of a classic MCTS tree before expansion.

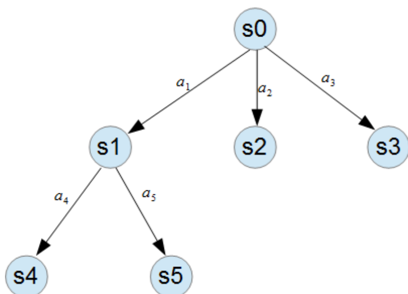


Figure 4: Example of a classic MCTS tree after expansion.

The tree is the only data structure used by the algorithm and it plays a central role during the search. When compared to A*, leaves of the tree roughly correspond to the open list and inner nodes to the closed list. Evaluation is done by averaging results of simulations instead of using a heuristic estimator. Similarly to A*, MCTS also extends promising paths by expanding appropriate nodes. Node selection process, however, works sequentially and it's quite different than that used by A*.

In the next section we explain an *enhanced MCTS* which uses simple planning algorithms in the simulation phase. It is important to note that it is the MCTS algorithm that actually searches for good solutions. The low-level planners are just a tool it uses to speed-up the convergence. In other words, the (Enhanced) MCTS does much more than just combine low-level planners.

Enhanced MCTS

We enhance the MCTS tree in a following manner: to every leaf node, we add new successors - one for each low-level planner in the portfolio. We will call them *virtual leaves*. The selection phase will work exactly the same way and select the most urgent *virtual leaf* - which means that it selects the (real) leaf and then an algorithm to use. During the simulation phase, the selected algorithm will be used as a simulation policy. After the expansion, however, the virtual leaves will not remain in the tree as inner nodes, but will move to the new leaves instead.

In the figure 5, there is an example of an enhanced MCTS tree. s_0 is the initial state, s_1 to s_3 are other states. a_1 to a_3 are actions and Alg_1 to Alg_3 are virtual leaves, Alg_2 of s_2 is the selected leaf. Now the algorithm Alg_2 will run trying to find a plan with s_2 as its initial state. After reaching the goal state, the cost of the plan found is used in the back-propagation phase to update statistics of the nodes. In the figure 6, there is the tree after expansion. New states that are reachable from s_2 are added, but virtual leaves are not kept as inner nodes in the tree. The are copied to the successors together with all statistical information they were holding.

Information stored in the nodes involves number of simulations, sum of all previous simulation results and sum of squares of results (which is sufficient to compute the *urgency* of the node during the selection phase). Virtual leaves are copied to each of the newly added successors and still stores all the information. Therefore low-level planners that proved to by efficient before the expansion will still be preferred in the new leaves.

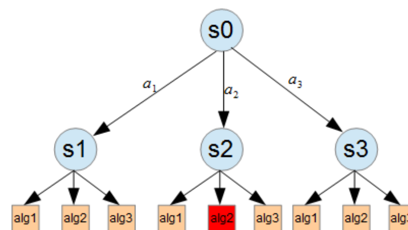


Figure 5: Example of an enhanced MCTS tree before expansion.

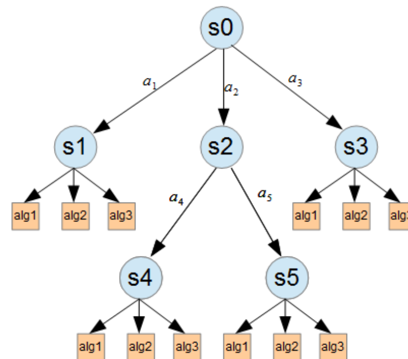


Figure 6: Example of an enhanced MCTS tree after expansion.

This way, only the real nodes remain in the tree (therefore saving space), but the algorithm is still able to use different search techniques in different parts of the tree. Inner nodes will accumulate all the simulation results no matter of the low-level algorithm that was used. This behavior is desired, since the simulation should be random and the results of any of the low-level algorithms could theoretically by generated by a random walk so it makes sense to accumulate the results.

This design allows us to overcome the problems with simulations when using MCTS for planning by using standard

planning algorithms in the simulation phase. It also creates a new hyper-heuristic framework for planning. The algorithm combines the *search for solutions* with the *search for search strategy* and it is continuously learning from its previous attempts.

From another point of view, the algorithm may be seen as a portfolio planner which uses MCTS to manage allocation of CPU time to individual low-level planners. The problem of allocating resources to individual algorithms in the portfolio is intensively studied and several approaches have been developed (Borrajo and Linares López 2012). Using MCTS for this task has certain advantages as the *MCTS selection* is proved to be optimal in a sense that the *regret* from not using the best algorithm all the time is asymptotically minimal (Auer, Cesa-Bianchi, and Fischer 2002). These attractive theoretical properties of MCTS selection make it a good choice for managing low-level planners.

There is an issue of redundant work when applying low-level planners to the selected leaf nodes. Planners might be solving very similar planning tasks over and over again, because states before an expansion and after it doesn't differ much. When using fast and simple planners, this should not be a problem. Moreover, we *need* more than one solution for each node in order to compute averages and other statistics that MCTS uses. In general, the MCTS approach is suitable for planning problems with large number of sub-optimal solutions that are relatively easy to find.

There might be difficulties when more sophisticated low-level planners are required due to the difficulty of the problem. Some redundant work can be eliminated by the use *AMAF* strategy (Helmbold and Parker-Wood 2009), but for hard problems where there is only very few goal states (or just one) the MCTS approach doesn't seem to be a good choice.

Beyond algorithm selection

Our design goes beyond simple *algorithm selection* as it allows different algorithms to be used in different parts of the search space. Consider the following example: the task is to solve a TSP with a slight modification: before the driver can start the tour, he has to unlock the car by solving a Rubik's cube. It is a standard planning problem that can be solved by any classical planner. To solve it, one would have to solve the Rubik's cube and then the TSP. Since these two problems are quite different, we might argue that they would be best solved by different algorithms.

An algorithm selection approach, as well as many portfolio planners would determine the best algorithm in this situation and then use it to solve the problem. No matter of the procedure to select the algorithm, the whole problem would in this case be solved by a single algorithm. Our approach on the contrary allows different algorithms to be used for the two parts of the problem. Initially, algorithms suited for Rubik's cube would be preferred during the simulation phase and the tree would grow towards good solutions for Rubik's cube. Once the tree grows enough to encompass the whole path to the solution of Rubik's cube, different algorithms would be preferred in the simulation phase, namely those well suited for the TSP problem.

To encourage such behavior, we slightly modify the update process for virtual leaves. After each expansion, when the virtual leaves are copied to successors, we multiply their current statistics by a number slightly less than 1. This makes recent simulations more important than the old ones, allowing the algorithm to adapt faster.

Although the mentioned example is quite artificial, there might be realistic domains that are composed of several different types of problems. On such domains, our approach would have a distinct advantage of being able to adapt itself to different aspects of the problem.

Algorithms for the simulation phase

In the simulation phase, we prefer speed over quality. To guarantee that simulations will be very fast, we set a limit on the length of the simulation. We propose the following search algorithms to be used in the simulation phase:

random walk - an ordinary random walk in the search space with a limit on its length. The search may end either by reaching a goal state or reaching the limit on length.

greedy hill-climbing with a heuristic - a standard hill-climbing, starts in the initial state and proceeds to the best successor according to the heuristic. The search ends when a goal state is reached or no successor is better than current state. Whenever there are more successors with best evaluation, one of them is chosen randomly.

f-limited A* - an A* with a limit F such that only nodes with f-value less or equal to F are added to the open list. The search ends when a goal state is reached or the open list empties. If the search fails to find a goal, the last expanded node is considered to be the result of the simulation.

beam search with narrow width - an A* where only a fixed number of best successors of selected node is added to the open list. To keep the process fast, we use the beam width of 2, meaning that only two best successors will be added each time. If there are more successors of the same best value, two of them will be chosen randomly. Again, the search ends when a goal state is reached or the open list empties and if the search fails to find a goal, the last expanded node is considered to be the result of the simulation.

planners from the* Agile track *of IPC - the Agile track focuses on finding suboptimal plans quickly which is exactly what we need. We believe some of the planners from this track could be used here, possibly with some modifications, but we have not been able to test this assumption experimentally yet.

Heuristic estimators So far, we have only tried the *FFheuristic*, weighted *FFheuristic*, heuristic counting the number of not accomplished goals and *blind heuristic* but we believe it is important to add more and especially to diversify better. There should be more types of heuristics in the pool to cover both symbolic and numerical types of problems.

There is also an issue of randomness in the simulations. The simulation phase is the only source of randomness in the

algorithm as all the other phases are completely deterministic. If we used only deterministic algorithms in the simulation phase, the whole system would be deterministic which might have its pros and cons. We decided to keep randomness in the design by using randomized search algorithms.

Evaluating simulations

In the simulation phase, we use fast and simple algorithms which may cause that many simulations will not be able to reach a goal state. We have developed a way to evaluate such simulations so that the algorithm may still gain some information from them. The evaluation criteria should meet the following requirements:

- simulations reaching a goal should be evaluated according to the cost of the plan
- simulations reaching dead end or dead component should be penalized for leading to unpromising parts of the space
- simulations ending due to the cut-off limit should be penalized for not reaching the goal
- simulations reaching a state close to goal should have better evaluation than those ending up far from any goal state

These seem very close to the A* selection criterion so that was our first choice. To evaluate the simulation that ended in a non-goal state, we used the A* criterion $f = g + h$, where f is the value of the simulation, g is the length of the path that the simulation took (i.e. sum of costs of actions used) and h would be a heuristic estimate of its final state. This criterion, however, doesn't work well. It prefers *shallow non-recognized dead ends* where by *shallow* we mean close to the initial state. Reasons for such behavior can be easily explained: by reaching a dead end, the simulation ends immediately so it is short (especially if the dead end is shallow) and has low g value. If the heuristic doesn't recognize the dead end, it assigns it some value less than infinity so it might get a reasonable h value. In that case, the sum $f = g + h$ is low and the search is drawn towards such unpromising states.

To overcome this problem, we slightly adjusted the evaluation criterion. For simulations that reach a goal state, we use simply the g-value (sum of costs of actions, h would in this case be zero anyway). For others that ended either by reaching a dead end or due to the limit, we use $f = g + h + (g - w)^2$, where g and h are the same as before and w is an estimate of the optimal plan length. This penalizes non-goal simulations by h value and furthermore, simulations that are too short or too long are penalized for being far from the *correct* length. This prefers states that are close to optimal goals and thanks to h, it should lead the search toward goal states. Although this works relatively well in our experiments, it might not be the best approach in general. The $(g - w)^2$ component is much larger than the other two which might be a problem on some domains. This issue still needs to be properly resolved in the future.

This approach eliminates shallow dead ends, but it requires the value w to be set. The value doesn't have to be precise - any reasonable overestimate will do. In most cases, we can easily come up with some rough estimate. In the experiments, we used $w = 200$ which worked fine for those problems. For domain-independent planning, however, the estimate should be done automatically. There are several ways to do that which we will explore in the future. One of them is to use a heuristic estimate of the initial state, possibly weighted by a constant between 1 and 10 to get an overestimate. Another option is to start with a small value of w and successively increase it until some goal states can are found.

In our experiments, we used the value w also as the limit on the length of simulations.

Design summary

Our design combines several simple search algorithms to find a close-to-optimal solution to an optimization planning task. It is capable of automatically selecting the best combination of search algorithms based on quality of solutions they found so far. The system is based on a hyper-heuristic principle and is able to adapt itself to different types of problem instances and even to different sub-problems within a single problem instance.

The search is guided by modified MCTS algorithm that searches for promising areas of the search space as well as for promising search algorithms. The *selection process* of MCTS guarantees asymptotically optimal distribution of computational effort between different areas of search-space and between different algorithms.

The hyper-algorithm itself is deterministic, the only source of randomness is through the low-level algorithms used during the simulation phase. We use stochastic algorithms, but deterministic simulation policies are also possible.

The overall algorithm is *asymptotically complete and optimal* in a sense that the probability of finding an optimal solution converges to 1 as the number of steps goes to infinity. This follows directly from the fact that the tree is evergrowing and the selection process guarantees that no branch will be skipped infinitely many times in a row. This is just a theoretical property though, and it tells us nothing about the speed of convergence.

The presented hyper-algorithm is independent of the low-level algorithms used in the simulation phase. That allows us to make use of any planning technique (both heuristic estimators and search algorithms) that has been developed so far or will be developed in the future.

Experiments

Our work is still in the beginning. So far, we have only performed a few small scale experiments to determine whether our design is usable at all. Larger-scale experiments with more domains comparing our system with others are necessary to properly asses the design. So far, we have conducted experiments to asses the price/performance ratio of the hyper-heuristic i.e. to discover whether the combination of simple algorithms is more powerful than the simple algorithms themselves. Results of the experiments suggest that the hyper-heuristic even with its overhead is worth using.

We performed experiments on two domains - *NoMystery* and *ZenoTravel*. We took several smaller problems from each domain and run planners for 5 minutes on each problem. We measured the quality of solution found. As the planners to be tested, we used *greedy hill-climbing* (HC), *f-limited A** (A*), *random walks with a length limit* (RW), *beam-search with beam width of 2* (BS) and a hyper-heuristic that uses the previous as its low-level algorithms (H-H). All informed algorithms use the FFHeuristic. The Table 1 shows the results. The numbers show quality of the plan found, *N/A* means that no plan has been found within the time limit. We add the length of the optimal plan (OPT) for comparison.

Table 1: Results of experiments with hyper-heuristic

Problem	A*	HC	RW	BS	OPT	H-H
mystery1	19	N/A	N/A	N/A	18	18
mystery2	24	N/A	N/A	N/A	19	22
mystery3	N/A	N/A	N/A	N/A	24	28
zeno2	6	6	6	6	6	6
zeno3	10	9	8	10	6	6
zeno4	11	10	13	10	8	9
zeno5	12	11	16	11	11	11
zeno6	15	14	26	14	11	12
zeno7	16	N/A	20	18	15	15
zeno8	16	16	32	N/A	11	13
zeno9	27	29	92	N/A	21	24

Conclusions and future work

We presented a hyper-heuristic domain-independent planner based on the MCTS algorithm. We described its properties, strengths and weaknesses and showed results of some preliminary experiments. Our main contributions can be summarized as follows:

- exploring the use of hyper-heuristic principle in classical planning
- using MCTS as a hyper-heuristic framework
- developing a new way of combining different search algorithms and heuristic estimators

Our approach is well suited for numerical planning domains where the number of goal states is high and they are easy to find and optimization is the real issue (e.g. transportation domains derived from TSP) as shown in the experiments. We believe that by using appropriate low-level search algorithms and heuristics, the system will be able to solve any planning task efficiently, including symbolic domains with very few goal states.

There are many aspects of the design that need to be explored further, especially deeper experimental analysis of behavior of the algorithm on various types of domains (what low-level algorithms does it use, what is the quality of solutions they find, how often are different low-level algorithms used in different parts of the search space and so on). There is also a lot of minor details that remain to be explored. For example: low-level algorithms use heuristics to guide the simulations and the MCTS then uses another heuristic to evaluate the results of those simulations. The question is whether we should use the same heuristic in both situations or two different heuristics will perform better and why.

It should also be interesting to study whether the *algorithm space landscape* really shows better properties (with respect to locality and ruggedness) then the *solution space* as it should, and what effect does such transformation have on different domains.

We believe that the hyper-heuristic principle has a great potential to be used in classical planning and that the idea should be explored further.

Acknowledgement

The research is supported by the Grant Agency of Charles University under contract no. 390214 and it is also supported by SVV project number 260 104.

We would like to thank the anonymous reviewers for their useful comments and suggestions.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* 47(2-3):235–256.

Baudiš, P. 2011. Balancing mcts by dynamically adjusting komi value. *ICGA Journal* 34:131–139.

Borrajo, D., and Linares López, C. 2012. Performance analysis of planning portfolios. In *Proceedings of the 5th Annual Symposium on Combinatorial Search (SOCS12)*. AAAI Press.

Burke, E. K.; Gendreau, M.; Hyde, M.; Kendall, G.; Ochoa, G.; Ozcan, E.; and Qu, R. 2013. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society* 64(12):1695–1724.

Cenamor, I.; de la Rosa, T.; and Fernandez, F. 2013. Learning predictive models to configure planning portfolios. In *Proceedings of the 4th workshop on Planning and Learning (ICAPS-PAL 2013)*, 14–22.

Chaslot, G.; Bakkes, S.; Szita, I.; and Spronck, P. 2008. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the 4th Artificial Intelligence for Interactive Digital Entertainment conference (AIIDE)*, 216–217. AAAI Press.

Fawcett, C.; Vallati, M.; Hutter, F.; Hoffmann, J.; Hoos, H. H.; and Leyton-Brown, K. 2014. Improved features for runtime prediction of domain-independent planners. In *24th International Conference on Automated Planning and Scheduling*.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Amsterdam: Morgan Kaufmann Publishers.

Hamadi, Y.; Monfroy, E.; and Saubion, F. 2012. *Autonomous search*. Springer-Verlag.

Helmbold, D. P., and Parker-Wood, A. 2009. All-moves-as-first heuristics in monte-carlo go. In *Proceedings of the 2009*

International Conference on Artificial Intelligence (ICAI09), 605–610.

Kazík, O.; Pešková, K.; Pilát, M.; and Neruda, R. 2011. Meta learning in multi-agent systems for data mining. In *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 02*, WI-IAT '11, 433–434. Washington, DC, USA: IEEE Computer Society.

Kocsis, L., and Szepesvári, C. 2006. Bandit based monte-carlo planning. In *Proceedings of the 15th European Conference on Machine Learning (ECML)*, 283–293. Springer Verlag.

Kumar, R.; Singh, S. K.; and Kumar, V. 2015. A heuristic approach for search engine selection in meta-search engine. In *Computing, Communication Automation (ICCCA), 2015 International Conference on*, 865–869.

Schadd, M. P. D.; Winands, M. H. M.; van den Herik, H. J.; Chaslot, G. M. J.-B.; and Uiterwijk, J. W. H. M. 2008. Single-player monte-carlo tree search. In *Proceedings of the 6th international conference on Computers and Games (CG '08)*, volume 5131 of *LNCS*, 1–12. Springer Verlag.

Thornton, C.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2013. Auto-weka: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, 847–855. New York, NY, USA: ACM.

Trunda, O., and Barták, R. 2013. Using monte carlo tree search to solve planning problems in transportation domains. In Castro, F.; Gelbukh, A. F.; and Gonzlez, M., eds., *MICAI (2)*, volume 8266 of *Lecture Notes in Computer Science*, 435–449. Springer.

Trunda, O. 2013. Monte carlo techniques in planning. Master's thesis, Faculty of Mathematics and Physics, Charles University in Prague.

Vallati, M.; Chrpa, L.; and Kitchin, D. E. 2014. Asap: An automatic algorithm selection approach for planning. *International Journal on Artificial Intelligence Tools* 23(6).