The 2014 International Planning Competition

Description of Participating Planners

Deterministic Track

The Eighth International Planning Competition
Description of Participant Planners of the Deterministic Track
June, 2014

Mauro Vallati m.vallati@hud.ac.uk
Lukáš Chrpa l.chrpa@hud.ac.uk
Thomas L. McCluskey t.l.mccluskey@hud.ac.uk

Planning, Autonomy and Representation of Knowledge group (PARK)
University of Huddersfield
Queensgate, Huddersfield
HD1 3DH, West Yorkshire
United Kingdom

Preface

This booklet summarizes the participants on the Deterministic Track of the International Planning Competition (IPC) 2014. Papers describing all the participating planners are included.
After a 3 years gap, the 2014 edition of the IPC involved a total of 67 planners, some of them versions of the same planner, distributed among four tracks: the sequential satisficing track (20 planners submitted out of 43 registered), the sequential multicore track (9 planners submitted out of 17 registered), the sequential optimal track (17 planners submitted out of 34 registered), the sequential agile track (15 planners submitted out of 21 registered) and the temporal satisficing track (6 planners submitted out of 9 registered). Three more tracks were open to participation: temporal optimal, preferences satisficing and preferences optimal. Unfortunately the number of submitted planners (at most 2 planners in each track) did not allow these tracks to be finally included in the competition.
A total of 66 people were participating, grouped in 31 teams. Participants came from Australia, Canada, Czech Republic, Finland, France, Germany, Iran, Israel, New Zealand, Spain, Switzerland, United Kingdom, Venezuela, USA.
For the sequential tracks 14 domains, with 20 problems each, were selected, while the temporal one had 10 domains, also with 20 problems each. Both new and past domains were included. As in previous competitions, domains and problems were unknown for participants and all the experimentation was carried out by the organizers.
To run the competition a cluster of 256 cores (AMD 2.39 Ghz QuadCore) using Linux was set up. Up to 1800 seconds, 4 GB of RAM memory and 200 GB of hard disk were available for each planner to solve a problem.
This resulted in 7400 computing hours (about 309 days), plus a high number of hours devoted to preliminary experimentation with new domains, reruns and bugs fixing.
The detailed results of the competition, the software used for automating most tasks, the source code of all the participating planners and the description of domains and problems can be found at the competition's web page:

http://helios.hud.ac.uk/scommv/IPC-14

Huddersfield, United Kingdom, June 2014

Mauro Vallati, Lukáš Chrpa, Thomas L. McCluskey
The IPC 2014, deterministic track, organisers

Acknowledgement

We want to thank all the people that submitted a planner to the deterministic track of the Eighth International Planning Competition.
Also, to all of you that suggested a domain to be included in the tracks, even if some were not accepted: **Patrik Haslum** submitted the *GED* domain; **Tomàs de la Rosa** and **Raquel Fuentetaja** sent us the *Pizza* and *Childsnack* domains; **Jörg Hoffmann** provided the *Crisp* domain; **Jussi Rintanen** is behind the *Maintenance* domain; **Jaanus Piip** and **Juhan Ernits** submitted the *Nurse Rostering* domain; **Héctor Luis Palacios** prepared a large number of conformant domains, such as G*rid, Cube, Emptyroom* and *Bomb*; **Nathan Robinson**, **Christian Muise** and **Charles Gretton** sent us the *Cave Diving* domain; **William Westerman** for the *Airport* domain and, last but not least, **Simon Parkinson** provided the domains *Calibration* and *Uncertainity*.
We do also want to thank **Daniel L. Kovacs** for making available a couple of manuscripts with a formal specification of PDDL 3.1.
We do feel in debt with **Ibad Kureshi**, **John Brennan** and, in general, to the High Performance Computing Research Group (HPC) of the University of Huddersfield for their assistance in configuring and making available the DES system and, moreover, for their continuous support during the testing phase.
Very importantly as well, to the IPC council for providing extensive comments and offering a lot of helpful suggestions.
Our most sincere thanks to **Carlos Linares López** and **Sergio Jimenez Celorrio** for inviting us to their university, their assistance with so much insight and all the material produced at the previous IPC.
We also thank the **Sportsman Pub** in Huddersfield, which supported us with good beer and a comfortable place for taking important decisions.
Finally, we have to acknowledge the sponsorship of the **University of Huddersfield**.

Huddersfield, United Kingdom, June 2014

Mauro Vallati, Lukáš Chrpa, Thomas L. McCluskey
The IPC 2014, deterministic track, organisers

Index

1. Sequential Satisficing track

ArvandHerd 1
BFS(f) 6
BiFD 8
DAE-YAHSP 10
DPMPlan 13
Fast Downward Cedalion 17
Fast Downward Stone Soup 2014 28
Fast Downward Uniform 32
Freelunch 33
IBACOP 35
IBACOP2 35
Jasper 39
Mercury 43
MIPlan 13
NUCELAR 48
Planets 52
RPT 55
USE 61
YAHSP3 64
YAHSP3-MT 64

2. Sequential Agile track

ArvandHerd 1
BFS(f) 6
Fast Downward Cedalion 17
Freelunch 33
IBACOP 35
IBACOP2 35
Jasper 39
Madagascar 66
Madagascar-pC 66
Mercury 43
PROBE 6
SIW 6
USE 61
YAHSP3 64
YAHSP3-MT 64

3. Sequential Optimal track

AllPACA 71
cGamer 74
DPMPlan 13
Dynamic-Gamer 77
Fast Downward Cedalion 17
Gamer 77
hflow 85
h^{++} 87
h^{++}_{ce} 87
Metis 88
MIPlan 13
NUCELAR 48
RIDA 93
Rational Lazy A* 97
SPM&S 101
SymBA*-1 105
SymBA*-2 105

4. Sequential Multicore track

ArvandHerd 1
DAE-YAHSP 10
IBACOP 35
IBACOP2 35
MIPlan 13
NUCELAR 48
Planets 52
USE 61
YAHSP3-MT 64

5. Temporal Satisficing track

DAE-YAHSP 10
ITSAT 110
tBurton 118
Temporal Fast Downward 121
YAHSP3 64
YAHSP3-MT 64

ArvandHerd 2014

Richard Valenzano, Hootan Nakhost*, Martin Müller, Jonathan Schaeffer
University of Alberta
{valenzan, nakhost, mmueller, jonathan}@ualberta.ca

Nathan R. Sturtevant
University of Denver
sturtevant@cs.du.edu

Abstract

ArvandHerd is a sequential satisficing planner that uses a portfolio consisting of LAMA and Arvand. This planner won the multi-core track of the 2011 International Planning Competition. In this paper, we describe the various components of ArvandHerd, the updates made for the 2014 competition, and the modifications that allow ArvandHerd to compete in the single-core sequential satisficing tracks.

1 Introduction

In the 2011 International Planning Competition, the winner of the multi-core track was a planner called ArvandHerd. This planner uses a portfolio-based approach to combine the strengths of the complementary approaches of random-walk and best-first search based planning. This is accomplished by simultaneously using the LAMA (Richter and Westphal 2010) and Arvand (Nakhost and Müller 2009) planners.

An updated version of this planner has been submitted to the 2014 competition. ArvandHerd 2014 is very similar to the planner which competed in 2011 and uses the same code base. However, it has been updated in several ways. These updates include the addition of techniques to LAMA including ϵ-greedy node selection, aggressive restarting, and diverse any-time search. We have also modified the planner so that it could compete in the single-core sequential satisficing tracks. In this paper, we will briefly consider the various components of ArvandHerd, describe the newly added techniques, and look at how the standard multi-core version of this planner has been modified so that it can compete in the single-core tracks.

2 The Components of ArvandHerd

ArvandHerd is a portfolio-based planner that uses a variety of planning techniques. In this section, we will briefly describe those components that have remained mostly the same from the 2011 version of this planner.

2.1 The ArvandHerd Code Base

The version of ArvandHerd submitted to the 2011 competition ran multiple threads from a single C++ binary. This is because ArvandHerd used both Arvand and LAMA, and Arvand had been built on top of LAMA 2008. Though there have been updates to each of these planners in LAMA 2011 (Richter, Westphal, and Helmert 2011) and Arvand 2013 (Nakhost and Müller 2013), the version of ArvandHerd submitted to the 2014 competition is still based on the LAMA 2008 code base.

The only component of this code base which was updated is the PDDL to SAS+ (Bäckström and Nebel 1995) translator, and the knowledge compilation step needed for the landmark count heuristic (Richter, Westphal, and Helmert 2011) used in LAMA. In the 2011 competition, ArvandHerd used the translator and knowledge compilation code in LAMA 2008. These pieces crashed on some of the problems in the 2011 competition, and so we have used the translator and knowledge compilation code from the version of Fast Downward used in IPC 2011. For details on how this translation is performed see (Helmert 2009). Details on the knowledge compilation step can be found in (Helmert 2006).

2.2 LAMA

LAMA is the winner of both the 2008 and 2011 IPC competitions, and is therefore a natural candidate for use as the greedy best-first search planner included in the portfolio. This planner uses a number of different techniques including multiple heuristics, preferred operators, deferred heuristic evaluation, and Restarting Weighted A* (RWA*). For a more complete description of this planner, see (Richter and Westphal 2010).

For the 2011 competition, a few additions were made to LAMA for its use in ArvandHerd. In particular, the planner was set to use random operator ordering, to cache the heuristic values of states in between iterations of the RWA* search, and the planner was modified so that a single call for the computation of the FF heuristic could be used to return both the action-cost aware or action-cost unaware versions of this heuristic. ArvandHerd also added a memory usage estimator to LAMA. This system estimates how much memory LAMA is using, and it allows the search to be restarted whenever a given memory limit is reached so that another parameterization of LAMA can be tried. These additions were also used in the 2014 version of ArvandHerd. For more information regarding how LAMA is used in ArvandHerd see (Valenzano et al. 2012) and (Valenzano et al. 2011).

* Now at Google.

2.3 Arvand

Arvand is a random-walk based planner that has been shown to be effective in certain domains that are difficult for best-first search based planners (Nakhost and Müller 2009). The execution of Arvand consists of a series of *search episodes*. Each episode starts by performing a set of random walks from the initial state and using the heuristic function to evaluate the endpoint of each of these random walks. Once a state with a low heuristic value is found, or after a certain number of such walks, the search jumps to the end of the walk on which the best heuristic value was found. The search episode then continues with a set of random walks from this state. This process then repeats until either a goal is found, or enough jumps are made without heuristic progress, in which case the search starts with a new search episode from the initial state. For a more thorough description of Arvand see (Nakhost 2013), (Nakhost and Müller 2009), (Nakhost and Müller 2013), and (Nakhost, Hoffmann, and Müller 2012).

Arvand has a number of parameters that allow the user to control the length of the random walks, the frequency with which the algorithm jumps during a search episode, and the frequency with which a search episode is terminated and the algorithm restarts. Since different parameterizations of the algorithm are best for different problems, Arvand has been designed so that a single instance of this planner can use different configurations in different search episodes. In the version of Arvand used in the 2011 competition, a *configuration selection* system was used to determine the parameterization to use on the next search episode. This system, which is based on the idea of a multi-armed bandit algorithm and is also used in ArvandHerd 2014, biases Arvand to more frequently use those parameterizations which have previously made the most heuristic progress.

Arvand also uses a second technique for sharing information across search episodes. This feature, which is called a *walk pool* (Nakhost, Hoffmann, and Müller 2012), stores those search trajectories which made the most search progress. When starting a new search episode, these trajectories are used to suggest an alternative starting point for the episode that is deeper into the state-space than the initial state. For a more in-depth description of how Arvand uses these features in ArvandHerd see (Valenzano et al. 2012).

2.4 Plan Improvement

In the sequential satisficing and sequential multi-core tracks, planner evaluation is based on the quality of solutions found. As such, when competing in these tracks it is critical to use the time remaining after a first solution is found to find better solutions. ArvandHerd uses multiple techniques for improving solution quality, and in this section we describe those which remain mostly the same from the version of this planner that was submitted to IPC 2011.

Aras. ArvandHerd uses a plan post-processing system called Aras (Nakhost and Müller 2010). The execution of this system consists of two phases. The first is a linear scan of a given solution path that looks for actions that can be removed such that the remaining plan is still valid. The second phase involves the construction of a neighbourhood graph around the solution path using a combination of forward search and a backwards, regression-based search. This graph is built until the number of nodes it contains reaches a given node limit. A search is then performed which finds the shortest path in this neighbourhood graph from the initial state to a goal state.

Aras runs by iterating between these two phases until some time or memory limit is reached, such that the limit on the nodes in the neighbourhood graph is increased each time the neighbourhood graph phase begins. All solutions found by ArvandHerd using either LAMA or Arvand are fed to Aras in an effort to improve solution quality.

Restarting Weighted A* (RWA*). RWA* was a feature introduced in the original version of LAMA that was later analyzed in (Richter, Thayer, and Ruml 2010). When using this technique, LAMA restarts and begins a less greedy search from scratch each time a solution is found. For example, on the first iteration of LAMA, the planner uses a greedy best-first search for finding the first solution. On the second iteration, LAMA then runs WA* with a weight of 10. If a second solution is found, WA* is run again but with an even smaller weight. This process then repeats until the time limit is reached.

In the standard version of RWA*, no information is shared between the iterations of RWA* except for the best solution found thus far, and the heuristic values of nodes that have already been expanded. This means that the search will not consider any nodes whose g-cost is as large as the best solution found thus far. This was the approach taken by ArvandHerd in 2011, though we used a different technique in ArvandHerd 2014 as described in Section 3.3.

Any-Time Arvand. Arvand was also set to continuously look for solutions even after a first solution is found. This simply means that Arvand continues to perform search episodes. As in LAMA, the cost of previously found solutions were also used to bound the search. This means that episodes with a g-cost that is larger than the bound are forced to restart. However, unlike how LAMA was used in ArvandHerd 2011, the bound used for the search episodes is only based on the best solution found by Arvand. As a result, the solutions found by LAMA or Aras are not factored into how search episodes are bound. This type of bounding was employed because Arvand is often unable to find any new solutions if the bound is too tight. By restricting the bound to only consider solutions found by Arvand means that the bound is looser than it would be if the other solutions were also factored in. This often allows Arvand to produce more plans, thereby increasing the chance that a plan will be found that will be greatly improved by Aras.

While this would suggest that perhaps no bounding should be used, experimentation with this system did indicate that some bounding was useful in certain domains in which Aras was ineffective at improving the solution quality. Using the best solution found by Aras was experimentally found to be an effective compromise between finding enough solutions for Aras while still adding useful bounding for domains in which Aras was not as successful. Note

that similar behaviour has been seen when using LAMA (Xie, Valenzano, and Müller 2013), and so we consider bounding in LAMA in Section 3.3.

3 Additions to ArvandHerd 2014

In this section, we describe the main changes that have been made to ArvandHerd for its submission to IPC 2014. Note that several of these techniques require parameters to be set, and we will describe the parameter values used in each track in Section 4.

3.1 ϵ-Greedy Node Selection

ϵ-greedy node selection is a simple technique that effectively introduces random exploration into the search (Valenzano et al. 2014). This technique, which requires the user to set a parameter ϵ in the range from 0 to 1, works as follows. With probability $1 - \epsilon$, the search acts exactly as the search algorithm ordinarily would. For example, if the search being used is GBFS, then with probability $1 - \epsilon$ the search will select the node from the open list with the smallest heuristic value as the next node to be expanded. However, with probability ϵ, the search is forced to use a different policy. Specifically, the search with select a node uniformly at random from amongst those in the open list. ϵ therefore determines how often the algorithm exploits heuristic information, and how often it explores.

Despite its simplicity, ϵ-greedy node selection has been shown to improve the coverage of planners like LAMA, even though this planner is already using multiple techniques for introducing variation into its search (Valenzano et al. 2014). However, we use ϵ-greedy node selection slightly differently in LAMA than as explained above. This is because LAMA uses $2k$ open lists where there are k heuristics in use, with k of the open lists holding all open nodes (each ordered by a different heuristic), and k open lists holding only those nodes achieved with a preferred operator (again, with each ordered by a different heuristic). For each node expansion, LAMA must first select one of the $2k$ open lists, and then using the corresponding heuristic to select a node from that open list. In our implementation of ϵ-greedy node selection for LAMA, we have left the open list selection mechanism the same, but have modified each open list to return a randomly selected node from that open list with probability ϵ. For example, if LAMA selects one of the preferred operator open lists as the next to be used and $\epsilon = 0.3$, then there is a 70% chance that the next node to be expanded will correspond to the node achieved using a preferred operator which has the lowest heuristic value and a 30% chance that the node will be randomly selected from the set of all nodes achieved using a preferred operator. This approach was taken due to the known effectiveness of the LAMA open list selection policy.

3.2 Aggressive Restarting

While the version of LAMA used in ArvandHerd in the 2011 competition would restart and use a different parameterization whenever the memory estimator indicated that a given memory limit was reached, an investigation that was performed after the competition suggested that this was not an effective restarting policy (Valenzano et al. 2012). In particular, if the search does not use up the memory quickly enough, it may spend all its time using an ineffective planner parameterization or an unlucky operator ordering. Moreover, if it does quickly use up the memory, the fact that it caches the heuristic values may mean that the other parameterizations do not have much memory to work with.

To remedy these problems, the restarting policy was modified in the 2014 version of ArvandHerd in two ways. In the first, we made the policy perform restarts much more often. This policy requires the user to set two parameters: an initial node expansion limit L_i and a limit factor L_f. The execution of LAMA in ArvandHerd 2014 begins with a node expansion limit of L_i. When this limit (or a memory limit) is hit, LAMA will restart and use a different configuration. Once the limit is reached with all of the configurations, the expansion limit is increased by a factor of L_f. This process then repeats until the time limit is reached.

So as to avoid the problem by which the additional parameterizations do not have enough memory for their search, we have set LAMA to clear its heuristic value cache if it reaches the memory limit too many times.

3.3 Diverse Any-Time Search

As mentioned above, if Arvand is set to bound its search episodes using the best solution found thus far including those from Aras, the bound often makes it too difficult to find any further plans. In that case, it was experimentally found to be better to use a looser bound so that Arvand finds more solutions and thus there is a greater chance that Aras will greatly improve at least one of them.

In (Xie, Valenzano, and Müller 2013) it was shown that similar behaviour was found when using Aras along with RWA* in LAMA. To remedy this situation, a new technique was developed called *Diverse Any-time Search (DAS)*. When using this approach, the planner runs RWA* as it typically does, but once a given time limit is hit, it starts a new RWA* search that begins again with the greediest of the configurations. This new RWA* search also ignores the cost of all previous solutions found. For example, if the time limit is five minutes, then a new RWA* search will begin again with GBFS every five minutes, and the bound used at any time is given by the best solution found during the current five minute RWA* phase. While Aras is also used on all solutions found, the cost of the solutions found by Aras are never used for bounding so as not to make it too difficult for LAMA to find new plans.

DAS was added to the RWA* search of LAMA in the version of ArvandHerd submitted to IPC 2014. The main difference with how it is described in (Xie, Valenzano, and Müller 2013) is how the RWA* time limit is set. In ArvandHerd 2014, we simply use the restarting policy described in the previous section to determine when a new DAS phase should begin. This means that a new DAS phase will begin once each configuration being used finds a solution or hits the current node expansion limit. The bound used during a DAS phase is given by the best solution found of all configurations tried with the same node expansion limit. Once all configurations have been tried with a particular

node expansion limit, the limit is increased according to the node limit factor, and a new RWA* search is started with a higher limit but no bound. Note that when a solution is found for a first time, we delay the increase of the expansion limit for one more RWA* phase.

4 Multi-Core and Single-Core ArvandHerd

While the version of ArvandHerd submitted in 2011 was solely a multi-core planner, the current version has been modified so that it can also compete in the sequential satisficing and the sequential agile tracks. In this section, we describe the differences between the versions used in these tracks including the parameterizations used.

4.1 Multi-Core ArvandHerd

The multi-core version of ArvandHerd runs almost identically to the way it did in 2011. From the single binary, ArvandHerd runs four threads. One of these threads runs LAMA while the other three run a parellelized version of Arvand. This parallel Arvand essentially has each thread run an independent search episode, although the threads share a single walk pool and a single configuration selector. For more information on this architecture, see the description of ArvandHerd given in (Valenzano et al. 2012). The only difference between the 2011 and 2014 versions of this system is that the Arvand threads no longer share the best solution they have found thus far with the thread running LAMA in the 2014 system. This is because, as described in Section 3.3, LAMA does not use the best solution found for bounding the search.

Multi-Core Parameters. In the multi-core version of ArvandHerd, there are four configurations made available for the configuration selector of Arvand. Two of these configurations bias the random walks to avoid using actions that have previously lead to dead-end states, while the other two bias the random walks to use helpful actions (Hoffmann and Nebel 2001) suggested by the heuristic. All configurations use a version of the FF heuristic (Hoffmann and Nebel 2001) which is not aware of action costs, but the configurations differ slightly in the initial length of the random walks, and how quickly the random walk length is increased. Multi-core ArvandHerd also uses a walk pool which holds a maximum of 100 search trajectories.

LAMA's RWA* has been set to run GBFS, then WA* with weights of 5 and 1. During the GBFS search, it uses a version of the FF heuristic that ignores action costs, while it uses a version which is aware of action costs when performing WA*. Both types of search also use the landmark count heuristic (Richter and Westphal 2010) and ϵ-greedy node selection with $\epsilon = 0.3$. Regarding the restart policy, the initial expansion limit is set at 100 while the node limit factor is set at 10.

4.2 Single-Core ArvandHerd

ArvandHerd did not compete in any single-core tracks in 2011, though it has been submitted to the sequential satisficing and sequential agile tracks in the 2014 competition. The single-core version of this planner does not use multiple threads. Instead, it runs Arvand first until a time limit is hit, and then it switches to LAMA. This is similar to the approach taken by Fast Downward Stone Soup (Helmert and Röger 2011), which also runs different planners in sequence.

Sequential Satisficing Parameters. In the sequential satisficing track, ArvandHerd runs Arvand for the first 15 minutes of the available runtime. The remaining time is then used by LAMA. The rest of the parameters are set just as they were for the multi-core track, except for the size of the walk pool which has been decreased to hold only a maximum of 50 trajectories.

Agile Satisficing Parameters. In the agile satisficing track, ArvandHerd runs Arvand for the first 3 minutes and LAMA for the remaining time. The parameters used are the same as in the sequential satisficing track, except that the walk pool size is decreased to hold a maximum of 20 trajectories, ϵ-greedy node selection is used with $\epsilon = 0.2$, and there is no weight 1 WA* configuration included in the set of LAMA configurations. Since solution quality is not counted in measuring performance in this track, Aras is not used, and ArvandHerd terminates once a first solution is found.

5 Conclusion

In this paper we have described the ArvandHerd planner submitted to the 2014 International Planning Competition. In particular, we have described the various components of this planner, the new techniques added since the 2011 competition, and how the planner has been made sequential for use in the single-core sequential satisficing tracks.

Acknowledgments

We would like to thank Fan Xie for the helpful discussions regarding techniques for improving plan quality in ArvandHerd 2014. This research was supported by GRAND and the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

Bäckström, C., and Nebel, B. 1995. Complexity Results for SAS+ Planning. *Computational Intelligence* 11:625–656.

García-Olaya, A.; Jiménez, S.; and López, C. L. 2011. The 2011 International Planning Competition. Technical report, Universidad Carlos III de Madrid, Madrid, Spain. http://hdl.handle.net/10016/11710.

Helmert, M., and Röger, G. 2011. Fast Downward Stone Soup: A Baseline for Building Planner Portfolios. In *The Proceedings of the 2011 ICAPS Workshop on Planning and Learning*, 28–35.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5-6):503–535.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14:253–302.

Nakhost, H., and Müller, M. 2009. Monte-Carlo Exploration for Deterministic Planning. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 1766–1771.

Nakhost, H., and Müller, M. 2010. Action Elimination and Plan Neighborhood Graph Search: Two Algorithms for Plan Improvement. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, 121–128.

Nakhost, H., and Müller, M. 2013. Towards a Second Generation Random Walk Planner: An Experimental Exploration. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*.

Nakhost, H.; Hoffmann, J.; and Müller, M. 2012. Resource-Constrained Planning: A Monte Carlo Random Walk Approach. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling*.

Nakhost, H. 2013. *Random Walk Planning: Theory, Practice, and Application*. Ph.D. Dissertation, University of Alberta. http://hdl.handle.net/10402/era.31939.

Richter, S., and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Richter, S.; Thayer, J. T.; and Ruml, W. 2010. The Joy of Forgetting: Faster Anytime Search via Restarting. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling*, 137–144.

Richter, S.; Westphal, M.; and Helmert, M. 2011. LAMA 2008 and 2011. In *The 2011 International Planning Competition* (2011) 50–54. http://hdl.handle.net/10016/11710.

Valenzano, R. A.; Nakhost, H.; Müller, M.; Schaeffer, J.; and Sturtevant, N. R. 2011. ArvandHerd: Parallel Planning with a Portfolio. In *The 2011 International Planning Competition* (2011) 113–116. http://hdl.handle.net/10016/11710.

Valenzano, R. A.; Nakhost, H.; Müller, M.; Schaeffer, J.; and Sturtevant, N. R. 2012. ArvandHerd: Parallel Planning with a Portfolio. In *20th European Conference on Artificial Intelligence*, 786–791.

Valenzano, R. A.; Sturtevant, N. R.; Schaeffer, J.; and Xie, F. 2014. A Comparison of Knowledge-Based GBFS Enhancements and Knowledge-Free Exploration. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*.

Xie, F.; Valenzano, R.; and Müller, M. 2013. Better Time Constrained Search via Randomization and Postprocessing. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, 269–277.

Width and Inference Based Planners: *SIW*, *BFS(f)*, and *PROBE*

Nir Lipovetzky
University of Melbourne
Melbourne, Australia
@unimelb.edu.au

Miquel Ramirez
RMIT University
Melbourne, Australia
@rmit.edu.au

Christian Muise
University of Melbourne
Melbourne, Australia
@unimelb.edu.au

Hector Geffner
ICREA & U. Pompeu Fabra
Barcelona, SPAIN
@upf.edu*

Introduction

We entered the planners *SIW*, $BFS(f)$, and $PROBE$ to the *agile-track* of the 2014 International Planning Competition, and an anytime planner for the *satisficing track* that runs both *SIW* and $BFS(f)$. SIW and $BFS(f)$ are classical planners that make use of the notion of width (Lipovetzky and Geffner 2012), while PROBE is a standard best-first search planner that augments the expansion of a node by throwing an "intelligent" probe which either reaches the goal or terminates quickly in low polynomial time (Lipovetzky and Geffner 2011). The basic building block of *SIW* is the Iterative Width Procedure (*IW*) for achieving one atomic goal at a time. *IW* runs in time exponential in the problem width by performing a sequence of pruned breadth first searches. The planner $BFS(f)$ integrates a *novelty* measure from *IW* with helpful-actions, landmarks and delete-relaxation heuristics in a Greedy Best-Fist search.

In the following sections we introduce the basic notions of the algorithms and the implementation.

SIW: Iterated Width Search

The algorithm *Iterated Width*, or *IW*, consists of a sequence of calls *IW*(i) for $i = 0, 1, \dots, |F|$ until the problem is solved. Each iteration *IW*(i) is a breadth-first search that prunes right away states that do not pass a *novelty* test; namely, for a state s in *IW*(i) *not* to be pruned there must be a tuple t of at most i atoms such that s is the first state generated in the search that makes t true. The time complexities of *IW*(i) and *IW* are $O(n^i)$ and $O(n^w)$ respectively where n is $|F|$ and w is the problem width. The width of existing domains is low for atomic goals, and indeed, 89% of the benchmarks can be solved by *IW*(2) when the goal is set to any one of the atoms in the goal (Lipovetzky and Geffner 2012). The width of the benchmark domains with conjunctive goals, however, is not low in general, yet such problems can be serialized.

The algorithm *Serialized Iterative Width*, or *SIW*, uses *IW* for serializing a problem into subproblems and for solving the subproblems. SIW uses IW to greedily achieve one atomic goal at a time until all atomic goals are achieved jointly. In between, atomic goals may be undone, but after each invocation of IW, each of the previously achieved goals must hold. SIW will thus never call IW more than $|G|$ times where $|G|$ is the number of atomic goals. SIW compares surprisingly well to a baseline heuristic search planner based on greedy best-first search and the h_{add} heuristic (Bonet and Geffner 2001), but does not approach the performance of the most recent planners. Nonetheless, *SIW* competes well in domains with no dead-ends and simple serializations.

BFS(f): Novelty Best-First Search

While the blind-search *SIW* procedure competes well with a greedy best-first planner using the additive heuristic, neither planner is state-of-the-art. For this, we developed a standard *forward-search best-first planner* guided by an evaluation function that combines the notions of novelty and helpful actions (Lipovetzky and Geffner 2012; Hoffmann and Nebel 2001). In this planner, called *BFS(f)* (Lipovetzky and Geffner 2012), ties are broken lexicographically by two other measures: (1) the number of subgoals not yet achieved up to a node in the search, and (2) the additive heuristic, h_{add}. The additive heuristic is delayed for non-helpful actions.

PROBE

PROBE is a complete, standard greedy best first search (GBFS) STRIPS planner using the standard additive heuristic (Bonet and Geffner 2001), with just *one change*: when a state is selected for expansion, it first launches a *probe* from the state to the goal (Lipovetzky and Geffner 2011). If the probe reaches the goal, the problem is solved and the solution is returned. Otherwise, the states expanded by probe are added to the open list, and control returns to the GBFS loop. The crucial and only novel part in the planning algorithm is the definition and computation of the probes.

PROBE is built using an early-version of an automated planning toolkit that supports the implementation details of the width-based algorithms. The only difference with respect to PROBE-IPC7 is that the anytime procedure is disabled, as we are only concerned with the first solution.

---

*firstname.lastname

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Implementation Notes

The planners *SIW*, $BFS(f)$ have been implemented using the automated planning toolkit `lwaptk`[1]. The toolkit is an extensible C++ framework that decouples search and heuristic algorithms from PDDL parsing and grounding modules, by relying on planner "agnostic" data structures to represent (ground) fluents and actions. We consider `lwaptk` to be a valuable contribution in itself since it allows to develop, relying on a collection of readily available implementations of search algorithms and planning heuristics, planners which are independent from specific parsing modules and grounding algorithms. If the planner is to be acquiring descriptions of planning tasks from PDDL specifications, the toolkit provides the means to plug into the planner either FF (Hoffmann and Nebel 2001) or FAST-DOWNWARD (Helmert 2006) parsers. Alternatively, and more interestingly, the planner can be embedded into complex applications, *directly*, if the "host" application is written in C++, or *indirectly* when the host is written in an interpreted language, such as PYTHON, by wrapping the planner with suitably generated marshalling code.

Acknowledgments This work was partly supported by Australian Research Council Linkage grant LP11010015, and Discovery Projects DP120100332 and DP130102825.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129:5–33.

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

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

Lipovetzky, N., and Geffner, H. 2011. Searching for plans with carefully designed probes. In *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling (ICAPS 2011)*, 154–161.

Lipovetzky, N., and Geffner, H. 2012. Width and serialization of classical planning problems. In *Proceedings of the Twentieth European Conference on Artificial Intelligence (ECAI 2012)*, 540–545.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, 975–982.

[1] Source code available from `https://github.com/miquelramirez/lwaptk`.

BiFD: Bidirectional Fast Downward

Vidal Alcázar, Susana Fernández, Daniel Borrajo
Universidad Carlos III de Madrid
Av. Universidad, 30
28911 Leganés, Spain
valcazar@inf.uc3m.es;sfarregu@inf.uc3m.es;dborrajo@ia.uc3m.es

Abstract

Most domains used as benchmarks in automated planning are asymmetrical. In general, it is unclear whether searching forward or backward is the best option *a priori*. Furthermore, planners that search backward have important differences (both advantageous and disadvantageous) with those that search forward. Here we propose BiFD, which combines Fast Downward with FDr (Fast Downward Regression) in a portfolio. This aims to exploit the advantages of searching in both directions while trying to minimize their disadvantages.

Motivation and Implementation

Since heuristic search in planning was proposed (Bonet and Geffner 2001), the option of searching forward and backward in the space of states is possible. The version that searched forward was called HSP; the version that searched backward was called HSPr. HSP proved experimentally that on average it's better to search forward, and that heuristic search in the state space is an efficient approach in satisficing planning. Because of this, most researchers that worked on satisficing planning focused on forward state-space techniques, forgoing the path opened by HSPr.

After almost 15 years there's a much better understanding of heuristic search in satisficing planning. Very successful planning systems based on forward heuristic state-space search have been implemented, like FF (Hoffmann and Nebel 2001) and Fast Downward (Helmert 2006), and a big number of papers on the matter have been published. However, the set of domains used as benchmarks has also grown in size and diversity, and experimentation has shown that forward heuristic state-space search, while good, does not dominate other approaches.

Some domains are known to be hard for forward search. For example, in the last International Planning Competition the hardest domain overall was *Floortile*. This was to be expected, as most planners performed forward search and *Floortile* is a trap domain specifically tailored to be difficult for them. *Floortile*'s trap consists in the occurrence of numerous dead ends that are undetectable by a reachability analysis when exploring the state space forward (see Figure 1). However, other planners may not be affected - and indeed aren't. Newer versions of Madagascar (Rintanen 2012), which (kind of) perform regression after compiling the task

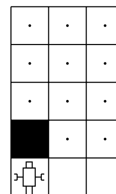


Figure 1: The goal in this instance of *Floortile* is to paint the dotted cells. The robot can only paint up and down; if, as in the example, the robot paints a cell before the cells above it are painted, the robot won't be able to paint at least one cell on that column. When this happens there is no way to solve the problem, which means that the depicted state is a dead end. This kind of dead ends are undetectable for reachability hueristics, but never happen in regression.

into a SAT instance, are able to easily solve the whole set of problems, and good ol' HSPr solves 18 out of 20 problems too. Other domains like *Storage* and *Matching-BW* were designed with the same intention in mind, and are in fact harder for planners like LAMA (Richter and Westphal 2010) than for Madagascar.

This was the main motivation for revisiting backward search in planning (Alcázar et al. 2013). In this paper several techniques initially proposed for progression were analyzed for their use in regression, which lead to the implementation of Fast Downward Regression (FDr), a spiritual successor of HSPr implemented on top of Fast Downward. While still worse on average than Fast Downward, results change if they are analyzed on a *per domain* basis.

This means that, instead of searching only in one direction, maybe it's better to search in both. The Madagascar+LAMA portfolio does perform really well (Rintanen 2012), so maybe this is the case too for a combination of Fast Downward and FDr. This way, BiFD performs a single preprocessing phase - which includes the computation of h^2 forward and backward (Haslum 2008) and getting rid of spurious operators (Alcázar et al. 2013), something that Fast Downward doesn't do - and then searches in both directions, expanding in the direction that has spent the least amount of time so far.

Since expanding and evaluating individual nodes is quite fast, BiFD will end up allocating half of the time to each direction. Although more elaborated strategies are possible, this should be good enough: the time needed to solve a planning task usually grows exponentially, so allocating half of the time means that only a few problems should be lost for that direction while getting most of the problems that can be solved by the opposite direction.

The settings of both planners are the following:

- Fast Downward uses Greedy Best-First Search with delayed evaluation, the FF heuristic and preferred operators.
- FDr uses Greedy Best-First Search with regular evaluation, the cached FF heuristic and disambiguation per state, but no preferred operators.
- The maximum time for computing h^2 and disambiguating operators is 300 seconds.

Note that we could have used other techniques forward too. For example, the configuration for forward search could have been LAMA's, but we preferred to use a simpler set of techniques (probably the closest thing to a good state-of-the-art baseline) to get a clearer picture.

Wait, This Isn't Bidirectional Search!

You're right, BiFD is a portfolio whose only synergy is the common preprocessing phase. For BiFD to be a proper bidirectional planner it should detect collision of frontiers and return the first solution when that happens. This requires dealing with subsumption of states, which has been recently analyzed (Alcázar, Fernández, and Borrajo 2014). However, results show that doing so in satisficing planning is just not worth it; many domains have plenty of symmetries and transpositions, and this affects greedy search algorithms in bidirectional search a lot, because they commit strongly to a subtree that is likely to be quite different from the subtree explored in the opposite direction. This means that collisions occur close to the goal in either direction in most problems and consequently there is little benefit in detecting the collision.

Note that this conclusion doesn't apply to the optimal case, as state-space search algorithms that can prove optimality must try all the permutations and symmetries below a given f threshold. For example, the newest bidirectional versions of Gamer (Kissmann and Edelkamp 2011), an optimal symbolic search planner, obtain overall good results (Torralba and Alcázar 2013).

Anytime Phase

Since quality matters in the competition, we have enabled an anytime phase that begins right after the first solution is found. It searches forward iteratively using the following configurations: Greedy Best First Search with delayed evaluation, a cost-sensitive version of the FF heuristic and preferred operators; Weighted $A^\star$ with regular evaluation, a cost-sensitive version of the FF heuristic, preferred operators and $w = 5, 3, 1$ sequentially; $A^\star$ with a cost-sensitive version of the FF heuristic and no preferred operators; and blind search.

These settings haven't been thoroughly tested, they were mainly just a combination of those of LAMA and our intuition.

Acknowledgements

This work has been partially supported by a FPI grant from the Spanish government associated to the MICINN project TIN2008-06701-C03-03, and it has also been supported by the project TIN2011-27652-C03-02.

References

Alcázar, V.; Borrajo, D.; Fernández, S.; and Fuentetaja, R. 2013. Revisiting regression in planning. In *International Joint Conference on Artificial Intelligence*, 2254–2260.

Alcázar, V.; Fernández, S.; and Borrajo, D. 2014. Analyzing the impact of partial states on duplicate detection and collision of frontiers. In *International Conference on Automated Planning and Scheduling*.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Haslum, P. 2008. Additive and reversed relaxed reachability heuristics revisited. *Proceedings of the 6th International Planning Competition*.

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

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

Kissmann, P., and Edelkamp, S. 2011. Improving cost-optimal domain-independent symbolic planning. In *AAAI Conference on Artificial Intelligence (AAAI)*, 992–997.

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

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

Torralba, Á., and Alcázar, V. 2013. Constrained symbolic search: On mutexes, BDD minimization and more. In *Symposium on Combinatorial Search (SoCS)*, 175–183.

Divide-and-Evolve: the Marriage of Descartes and Darwin

Johann Dreo **Pierre Savéant**
Thales Research & Technology
Palaiseau, France
first.last@thalesgroup.com

Marc Schoenauer
INRIA Saclay & LRI
Orsay, France
marc.schoenauer@inria.fr

Vincent Vidal
ONERA – DCSD
Toulouse, France
Vincent.Vidal@onera.fr

Abstract

DAE$_X$, the concrete implementation of the *Divide-and-Evolve* paradigm, is a domain-independent satisficing planning system based on Evolutionary Computation. The basic principle is to carry out a *Divide-and-Conquer* strategy driven by an evolutionary algorithm. The key components of DAE$_X$ are a state-based decomposition principle, an evolutionary algorithm to drive the optimization process, and an embedded planner X to solve the sub-problems. The release that has been submitted to the competition is DAE$_{\text{YAHSP}}$, the instantiation of DAE$_X$ with the heuristic forward search YAHSP planner. The marriage of DAE and YAHSP matches a clean role separation: YAHSP gets a few tries to find a solution quickly whereas DAE controls the optimization process.

Introduction

This section introduces the main principles of the satisficing planner DAE, referring to (Bibaï et al. 2010c) for a comprehensive presentation. DAE$_X$, the concrete implementation of the *Divide-and-Evolve* paradigm, is a domain-independent satisficing planning system based on Evolutionary Computation (Schoenauer, Savéant, and Vidal 2006). The basic principle is to carry out a *Divide-and-Conquer* strategy driven by an evolutionary algorithm. The algorithm is detailed in (Bibaï et al. 2010a) and compared with state-of-the-art planners.

Given a planning problem $P = \langle A, O, I, G \rangle$, where A denotes the set of atoms, O the set of actions, I the initial state, and G the goal state, DAE$_X$ searches the space of sequences of partial states $(s_i)_{i \in [0,n+1]}$, with $s_0 = I$ and $s_{n+1} = G$: DAE$_X$ looks for the sequence such that the plan σ obtained by compressing subplans σ_i found by some embedded planner X as solutions of $P_i = \langle A, O, \hat{s}_i, s_{i+1} \rangle_{i \in [0,n]}$ has the best possible quality (with $\hat{s_i}$ denoting the final state reached by applying σ_{i-1} from $\hat{s}_{i-1}$). Each intermediate state $(s_i)_{i \in [1,n]}$ is first seen as a set of goals and then completed as a new initial state for the next step by simply applying the plan found to reach it. In order to reduce the number of atoms used to describe these states, DAE relies on the admissible heuristic function h^1 (Haslum and Geffner 2000a): only the ones that are possibly true according to h^1 are considered. A diagram of the decomposition approach in DAE is depicted on figure 1.

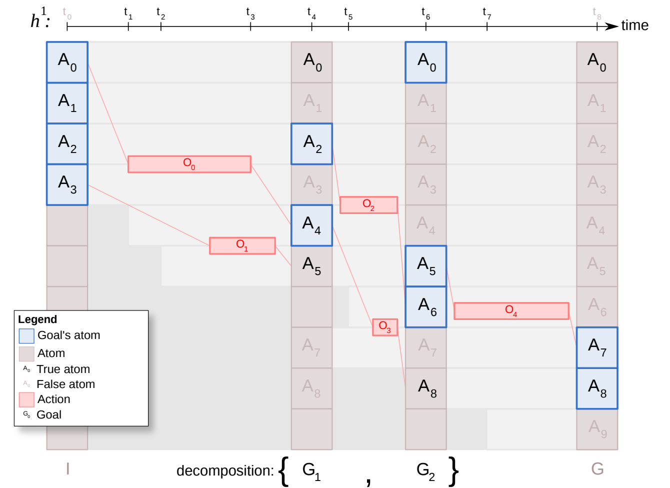


Figure 1: The decomposition approach used in DAE.

Furthermore, mutually exclusive atoms, which can be computed at low cost, are also forbidden in intermediate states s_i. These two rules are strictly imposed during the random initialization phase, and progressively relaxed during the search phase. The compression of subplans is required by temporal planning where actions can run concurrently: a simple concatenation would obviously not produce the minimal makespan.

Due to the weak structure of the search space (variable-length sequences of variable-length lists of atoms), Evolutionary Algorithms (EAs) have been chosen as the method of choice: EAs are metaheuristics that are flexible enough to explore such spaces, as long as they are provided with some stochastic *variation operators* (aka *move operators* in the heuristic search community) – and of course some objective function to optimize.

Variation operators in DAE are (i) a crossover operator, a straightforward adaptation of the standard one-point crossover to variable-length sequences; and (ii) different mutation operators, that modify the sequence at hand either at the sequence level, or at the state level, randomly adding or removing one item (state or atom).

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

The objective value is obtained by running the embedded planner on the successive subproblems. When the goal state is reached, a feasibility fitness is computed based on the compression of solution subplans, favoring quality; otherwise, an unfeasibility fitness is computed, implementing a gradient towards satisfiability (see (Bibaï et al. 2010c) for details).

DAE can embed any existing planner, and has to-date been successful with both the optimal planner CPT (Vidal and Geffner 2004) and the lookahead heuristic-based satisficing planner YAHSP (Vidal 2004). The latter has been demonstrated to outperform the former when used within DAE (Bibaï et al. 2010d), so only $\text{DAE}_{\text{YAHSP}}$ has been considered in this work.

The target is thus temporal satificing planning with conservative semantics, cost planning and classical STRIPS planning. The marriage of DAE and YAHSP matches a clean role separation: YAHSP gets a few tries to find a solution quickly whereas DAE controls the optimization process. In the current release we have introduced an initial estimation processing of the maximum number of tries allowed to YAHSP for all individual evaluations. This parameter is crucial for the time consumption of the algorithm.

Algorithms

DAE_{X} is an evolutionary algorithm, which basically mimics a biological evolution as a stochastic process (i.e. using biased random search in an iterative manner). Figure 2 depicts the main components of the evolution engine of $\text{DAE}_{\text{YAHSP}}$.

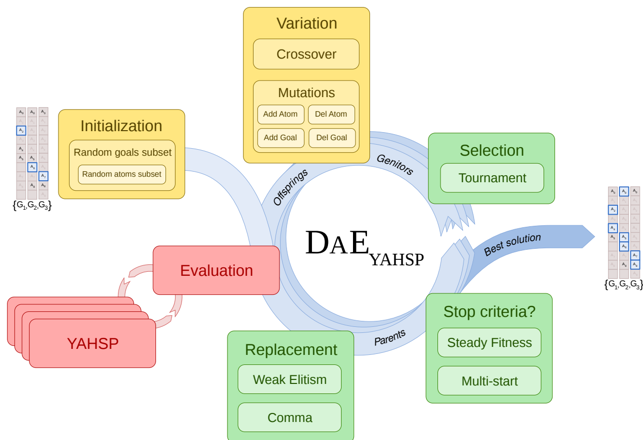


Figure 2: The evolution engine used in $\text{DAE}_{\text{YAHSP}}$. Yellow boxes indicates problem-dependent operators, green ones problem-independent operators and red boxes indicates the planner-dependent fitness evaluation. The output of the evolutionary algorithm is a decomposition of the problem.

The fitness implements a gradient towards feasibility for unfeasible individuals and a gradient towards optimality for feasible individuals. Feasible individuals are always preferred to unfeasible ones. Population initialization as well as variation operators are driven by the critical path h^1 heuristic (Haslum and Geffner 2000b) in order to discard inconsistent state orderings, and atom mutual exclusivity inference in order to discard inconsistent states. These two computations are done by YAHSP in an initial phase.

Beside a standard one-point crossover for variable length representations, four mutations have been defined: addition (resp. removal) of a goal in a sequence, addition (resp. removal) of an atom in a goal.

Variation operators relax the strictly h^1 ordering of atoms within individuals, since it is only a heuristic estimate.

The selection is a comparison-based deterministic tournament of size 5.

For the sequential release, Darwinian-related parameters of DAE_{X} have been fixed after some early experiments (Schoenauer, Savéant, and Vidal 2006) whereas parameters related to the variation operators have been tuned using the Racing method (Bibaï et al. 2010b). It should be noted that, due to the conditions of the competition, the parameter setting is global to all domains. In (Bibaï et al. 2010b) we showed that a specific tuning for an instance provides better results as expected and that what we would do for a real-life planning task.

We added two novelties to the version described in (Bibaï et al. 2010a). One important parameter is the maximum number of expanded nodes allowed to the YAHSP subsolver which defines empirically what is considered as an easy problem for YAHSP. As a matter of fact, the minimum number of required nodes varies from few nodes to thousands depending of the planning task. In the current release this number is estimated during the population initialization stage. An incremental loop is performed until the ratio of feasible individuals is over a given threshold or a maximum boundary has been reached. By default this number is doubled at each iteration until at least one feasible individual is produced or 100,000 has been reached.

Furthermore we add the capability to perform restarts within a time contract in order to increase solution quality.

The fitness used for the competition differs from the one described in (Bibaï et al. 2010a). The fitness for bad individuals has been simplified by withdrawing the Hamming distance to the goal. The new fitness depends only on the "decomposition distance": the number of intermediate goals reached and more specifically the one that are "useful". A useful intermediate goal is a goal that require a non-empty plan to be reached.

Implementation

The implementation of DAE_{X} has been made with the ParadisEO framework[1] which provides an abstract control structure to develop any kind of evolutionary algorithm in C++. YAHSP is written in the C language. The source code is available under an open-source license and the version used for the competition has the hash `9a46716` in the official repository[2].

In order to speed up search, a memoization mechanism has been introduced in YAHSP and carefully controlled to

[1] `http://paradiseo.gforge.inria.fr/`

[2] `https://gforge.inria.fr/git/paradiseo/paradiseo.git`

leave memory space for DAE. Indeed, most of the time during a run of YAHSP, and as a consequence during a run of $\text{DAE}_{\text{YAHSP}}$, is spent in computing the h^{add} heuristic for each encountered state (see (Vidal 2011) for more details about the algorithms of the new version of the YAHSP planner). During a single run of YAHSP, duplicate states are discarded; but during a run of $\text{DAE}_{\text{YAHSP}}$, the same state can be encountered multiple times. We therefore keep track of the h^{add} costs of all atoms in the problem for each state, in order to avoid recomputing these values each time a duplicate state is reached. This generally leads to a speedup comprised between 2 and 4. When $\text{DAE}_{\text{YAHSP}}$ runs out of memory, which obviously happens much faster with the memoization strategy, all stored states and associated costs are flushed. More sophisticated strategies may be implemented, e.g. flushing the oldest or less often encountered states; but we found that the simplest solution of completely freeing the memoized information was efficient enough.

Several biases have been introduced in YAHSP, in order to help $\text{DAE}_{\text{YAHSP}}$ finding better solutions. The main one is that actions of lower duration are preferred to break ties between several actions of same h^{add} cost, when computing relaxed plans and performing the relaxed plan repair strategy. Another bias is that the cost incrementation made during h^{add}, which is usually equal to 1 for each applied action, is made equal to either the duration or the cost of the action. Although these biases do not change a lot the quality of the plans produced by YAHSP alone, we found that they are of better help to $\text{DAE}_{\text{YAHSP}}$. However, introducing such biases is not very satisfactorily; it would be better to exactly use the version described in (Vidal 2011). We still have to better investigate the relationships between the evolutionary engine and the embedded planner, in order to determine how to manage such kind of biases and other tie-breaking strategies.

The version submitted to the sequential multi-core track use a parallelized evaluation operator that dispatch the fitness computation across multiple processes using message passing. No change is made to the $\text{DAE}_{\text{YAHSP}}$ algorithm, the implementation uses parallel operators wrappers available in the ParadisEO framework, with a static assignment of jobs. Note that while the source code permits a parallelization at the multi-starts level, it is not used in the competition.

Acknowledgments

This work is being partially funded by the French National Research Agency (ANR) through the COSINUS programme, under the research contract DESCARWIN (ANR-09-COSI-002).

References

Bibaï, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010a. An Evolutionary Metaheuristic Based on State Decomposition for Domain-Independent Satisficing Planning. In 20th *International Conference on Automated Planning and Scheduling (ICAPS-2010)*, 18–25. AAAI Press.

Bibaï, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010b. On the Generality of Parameter Tuning in Evolutionary Planning. In 20th *Genetic and Evolutionary Computation Conference (GECCO'10)*, 241–248. ACM Press.

Bibaï, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010c. An Evolutionary Metaheuristic Based on State Decomposition for Domain-Independent Satisficing Planning. In R. Brafman et al., ed., 20th *International Conference on Automated Planning and Scheduling (ICAPS-10)*, 18–25. AAAI Press.

Bibaï, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010d. On the Benefit of Sub-Optimality within the Divide-and-Evolve Scheme. In Cowling, P., and Merz, P., eds., *Proc.* 10th *EvoCOP*, 23–34. LNCS 6022, Springer Verlag.

Haslum, P., and Geffner, H. 2000a. Admissible Heuristics for Optimal Planning. In 5th *Int. Conf. on AI Planning and Scheduling (AIPS 2000)*, 140–149.

Haslum, P., and Geffner, H. 2000b. Admissible Heuristics for Optimal Planning. In *AIPS-2000*, 70–82.

Schoenauer, M.; Savéant, P.; and Vidal, V. 2006. Divide-and-Evolve: a New Memetic Scheme for Domain-Independent Temporal Planning. In Gottlieb, J., and Raidl, G., eds., 6th *European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP'06)*. Springer Verlag.

Vidal, V., and Geffner, H. 2004. Branching and Pruning: An Optimal Temporal POCL Planner Based on Constraint Programming. In *Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, 570–577. AAAI Press.

Vidal, V. 2004. A Lookahead Strategy for Heuristic Search Planning. In 14th *International Conference on Planning and Scheduling (ICAPS-04)*, 150–159. AAAI Press.

Vidal, V. 2011. YAHSP2: Keep It Simple, Stupid. In 7th *International Planning Competition (IPC-2011), Deterministic Part*.

MIPLAN and DPMPLAN

Sergio Núñez and **Daniel Borrajo** and **Carlos Linares López**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. 28911 Leganes (Madrid). Spain
sergio.nunez@uc3m.es, dborrajo@ia.uc3m.es, clinares@inf.uc3m.es

Abstract

In this document we describe the techniques used to configure the sequential portfolios submitted to the deterministic tracks of the International Planning Competition 2014. We have submitted four portfolios to sequential (optimal and satisficing) tracks and one to the multi-core track accounting for five different portfolios in total. All submitted portfolios have been configured using techniques based on Mixed-Integer Programming, which derive the best achievable performance with a linear combination of planners.

MIPLAN

MIPLAN portfolios have been generated using Mixed-Integer Programming (MIP), which computes the portfolio with the best achievable performance with respect to a selection of training planning tasks (Núñez, Borrajo, and Linares López 2012). The resulting portfolio is a linear combination of candidate planners defined as a sorted set of pairs <planner, time>. Our MIP model considers an *objective function* that maximizes a weighted sum of different parameters including overall running time and quality.

Since we consider two different criteria (time and quality), it could be viewed and solved as a multi-objective maximization problem. Instead, we solve two MIP tasks in sequence while preserving the cost of the objective function from the solution of the first MIP. Specifically, we first run the MIP task to optimize only *quality* —i. e., sum of the plan quality of each solved problem for the satisficing track and the total number of solved planning tasks for the optimal track. If a solution exists, then a second execution of the MIP model is performed to find the combination of candidate planners that achieves the same quality (denoted as Q) while minimizing the overall running time. To enforce a solution with the same quality an additional constraint is added: $\sum_{i=0}^{n} quality_i \geq Q - \epsilon$, where ϵ is just any small real value used to avoid floating-point errors. Clearly, a solution is guaranteed to exist here, since a first solution was already found in the previous step. Pseudocode 1 shows the steps followed to generate all the submitted portfolios where quality was maximized first, and then running time was minimized among the combinations that achieved the optimal quality. In our experiments, $\epsilon = 0.001$.

Algorithm 1 Build a portfolio optimizing quality and time

```
set weights to optimize only quality
portfolio_1 := solve the MIP task
Q := the resulting value of the objective function
if a solution exists then
    add constraint sum_{i=0}^{n} quality_i >= Q - 0.001
    set weights to optimize only overall running time
    portfolio_2 := solve the MIP task
    return portfolio_2
else
    exit with no solution
end if
```

The MIP task used in this work does not result in any particular order to execute the planners. It only assigns an execution time to each planner, which is either zero or a positive amount of time. The definition of the execution sequence is arbitrary and it is based just on the order in which the planners were initially specified.

As a matter of fact, it was empirically found that the MIP solver usually distributes all the available time among the candidate planners selected to be part of the portfolio. Running the procedure depicted in Pseudocode 1, the solution of the second MIP step could result in a sum of the times assigned to each planner that is less than the available time in the competition. Thus, it is possible to have some slack time which could be distributed uniformly among the selected planners. Besides, we can use this slack time to scripting tasks like checking if the current planner has solved the current problem and if so, validating the solution found by that component planner.

DPMPLAN

The technique used to configure MIPLAN portfolios is focused on maximizing the objective function at the last instant (measured in seconds) within the available time. For instance, if the available time is equal to 1800 seconds, the MIP task generates the portfolio with the best achievable performance in the instant equal to 1800 seconds.

The idea behind DPMPLAN is to modify the objective function used to configure MIPLAN in a temporal objective function. Thereby, the MIP task will maximize the objective

function for each instant (measured in seconds). However, this problem is too hard to solve. Therefore, instead of maximizing the objective function for each instant, we have only selected a few values: 1, 5, 10, 25, 50, 100, 200, 450, 900 and 1800 (time limit).

Implementation of the Portfolio

Every submitted portfolio runs a fixed portfolio configuration. However, the runtime assigned to each component planner can change in unexpected ways during its execution when the component planner finishes prematurely: planner bugs, terminating cleanly without solving the instance, running out of memory, etc. Therefore, the total runtime of the submitted portfolio can be lower than the available time. In this case, the submitted portfolio will run a default planner using the remaining time (RT). This default planner is picked up among the set of candidate planners which have not been selected to be part in the portfolio and had a remarkable performance in the IPC 2011.

Sequential Optimization Track

In this section, we briefly describe the experiments performed to configure MIPLAN and DPMPLAN for the sequential optimization track. We have applied both techniques over all the optimal planning tasks defined for the IPC 2011. Also, we have used all the planners considered in the design of FDSS (Helmert, Röger, and Karpas 2011) and all the participant planners in the IPC 2011, removing portfolios and adding their solvers instead. We have discarded FORKINIT, IFORKINIT and LMFORK because the organizers of the IPC 2014 had problems with the license of the Mosek LP solver.

Tables 1 and 2 show the configurations of the submitted portfolios. We have selected SELMAX as default planner for both MIPLAN and DPMPLAN portfolios.

Planner	Allotted time (s)
CPT4	20
BJOLP	70
LM-CUT	80
M&S-BISIM 1	160
M&S-BISIM 2	195
GAMER	1275
SELMAX	RT

Table 1: Configuration of MIPLAN for the sequential optimization track.

Planner	Allotted time (s)
CPT4	10
FD AUTOTUNE	25
RHW LANDMARKS	40
BJOLP	59
LM-CUT	65
M&S-BISIM 1	145
M&S-BISIM 2	180
GAMER	1276
SELMAX	RT

Table 2: Configuration of DPMPLAN for the sequential optimization track.

As we mentioned before, the MIP task does not specify the execution sequence of the generated portfolios. However, we have sorted the execution sequence of the submitted portfolios in increasing order of the allotted time.

Sequential Satisficing Track

Similarly to the previous track, we have applied both techniques over all the satisficing planning tasks chosen at the IPC 2011. However, the set of candidate planners is different for both portfolios. In the design of MIPLAN, we have considered all the participant planners in the IPC 2011 removing portfolios and adding their component solvers instead. The resulting portfolio is shown in Table 3, which shows that the default planner is ROAMER.

The implementation of the portfolio developed by the Fast-Downward planning system (Helmert 2006) allows the component planners to communicate information among them. However, their component planners must be defined using the Fast-Downward planning system. Therefore, given that we have considered the component planners of FDSS-1 and FDSS-2, we have built an auxiliary portfolio with the component planners of both FDSS portfolios selected by the MIP task using the implementation of Fast-Downward. Thereby, the component planners of this portfolio can communicate information among them. The configuration of this auxiliary portfolio, denoted as FD PORTFOLIO, is shown in Table 4.

Planner	Allotted time (s)
MADAGASCAR P	1
FD AUTOTUNE 1	5
YAHSP2 MT	5
YAHSP2	6
LAMAR	20
DAE YAHSP	29
LAMA 2008	40
ARVAND	49
PROBE	51
LAMA-2011	357
FD PORTFOLIO	585
FD AUTOTUNE 2	652
ROAMER	RT

Table 3: Configuration of MIPLAN for the sequential satisficing track.

In the design of DPMPLAN, we have considered all the participant planners in the IPC 2011. Table 5 shows the portfolio configuration of the submitted portfolio. The planner selected as default planner is LAMAR.

Planner			Allotted
Search	Evaluation	Heuristics	Time (s)
Greedy best-first	Eager	FF	48
Greedy best-first	Eager	FF, CG	59
Weighted-A* w=3	Eager	ADD	105
Greedy best-first	Eager	CG	142
Weighted-A* w=3	Lazy	CG	227
Greedy best-first	Lazy	CG	4

Table 4: FD PORTFOLIO of the MIPLAN portfolio for the sequential satisficing track.

Planner	Allotted time (s)
YAHSP2	5
CPT 4	6
MADAGASCAR P	10
YAHSP2 MT	10
DAE YAHSP	20
ARVAND	28
FDSS 2	63
FORKUNIFORM	66
LAMA 2008	78
LAMA 2011	90
FD AUTOTUNE 1	140
PROBE	160
ROAMER	472
FD AUTOTUNE 2	652
LAMAR	RT

Table 5: Configuration of DPMPLAN for the sequential satisficing track.

Sequential Multi-core Track

We have submitted one portfolio (MIPLAN) for the multi-core track. We have added the concept of core processor to the MIP model. Thus, the MIP task generates one portfolio configuration using the four cores available and respecting the wall-clock time limit set up in the competition.

We have used all the problems considered in the sequential satisficing track of the IPC 2011. Also, we have considered all the participant planners of this competition and the component solvers of the participant portfolios. The FDSS planners are defined by a search algorithm, an evaluation method and a set of heuristics. Specifically, FDSS only considered weighted-A* w=3 (WA*) and greedy best-first search (GBFS), with EAGER (standard) and LAZY (deferred evaluation) variants of both search algorithms. Also, only four heuristics were considered: additive heuristic ADD (Bonet and Geffner 2001), FF/additive heuristic FF (Hoffmann and Nebel 2001; Keyder and Geffner 2008), causal graph heuristic CG (Helmert 2004), and context-enhanced additive heuristic CEA (Helmert and Geffner 2008).

The configuration of the submitted portfolio is shown in Table 6, which shows for each component planner, its allotted time and the core id where the planner shall be executed. We have only selected two default planners with the purpose of using the memory available between two planners instead of four planners. In this case, the default planners are LAMAR and LAMA 2011. Although LAMA 2011 is a component planner of the submitted portfolio, we have selected it because as default planner has more available memory.

Planner	Core	Allotted Time
GBFS - LAZY - CG	0	25
WA* - EAGER - FF	1	612
FDSS 2	2	220
GBFS - EAGER - FF, CG	3	543
YAHSP2	0	25
YAHSP2 MT	0	25
MADAGASCAR-P	0	36
GBFS - EAGER - FF	0	40
GBFS - EAGER - CEA	0	42
GBFS - EAGER - ADD, FF, CEA	0	55
FD AUTOTUNE 2	2	1525
DAE YAHSP	0	79
WA* - EAGER - ADD	0	125
WA* - LAZY - FF	0	220
FORKUNIFORM	3	1232
ROAMER	1	618
LAMA 2011	0	320
GBFS - EAGER - ADD, FF, CG	0	222
LAMA 2008	0	225
WA* - LAZY - CG	1	570
PROBE	0	359
LAMAR	0	RT
LAMA 2011	1	RT

Table 6: Configuration of the MIPLAN portfolio for the multi-core track.

Acknowledgments

We automatically generated sequential portfolios of existing planners by means of a MIP task to be submitted to the International Planning Competition 2014. Thus, we would like to acknowledge and thank the authors of the individual planners for their contribution and hard work.

This work has been partially supported by the Planinteraction project TIN2011-27652-C03-02.

References

Bonet, B., and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence* 129(1-2):5–33.

Helmert, M., and Geffner, H. 2008. Unifying the Causal Graph and Additive Heuristics. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 140–147.

Helmert, M.; Röger, G.; and Karpas, E. 2011. Fast Downward Stone Soup: A Baseline for Building Planner Portfolios. *In ICAPS 2011 Workshop on Planning and Learning* 28–35.

Helmert, M. 2004. A Planning Heuristic Based on Causal Graph Analysis. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, 161–170. AAAI Press.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research (JAIR)* 14:253–302.

Keyder, E., and Geffner, H. 2008. Heuristics for Planning with Action Costs Revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, 588–592.

Núñez, S.; Borrajo, D.; and Linares López, C. 2012. Performance Analysis of Planning Portfolios. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS, Niagara Falls, Ontario, Canada, July 19-21, 2012*. AAAI Press.

Fast Downward Cedalion

Jendrik Seipp and **Silvan Sievers**
Universität Basel
Basel, Switzerland
{jendrik.seipp,silvan.sievers}@unibas.ch

Frank Hutter
Universität Freiburg
Freiburg, Germany
fh@informatik.uni-freiburg.de

To avoid duplication of content we only give a high-level overview of our algorithm here and refer to a technical report for details on our methodology (Seipp, Sievers, and Hutter 2013). An extended version of that report is forthcoming. The paper at hand focuses mostly on the configuration setup we used for generating portfolios for IPC 2014.

Portfolio Configuration

Cedalion is our algorithm for automatically configuring sequential planning portfolios. Given a parametrized planner and a set of training instances, it iteratively selects the pair of planner configuration and time slice that improves the current portfolio the most per time spent. At the end of each iteration all instances for which the current portfolio finds the best solution are removed from the training set. The algorithm stops when the the total runtime of the added configurations reaches the portfolio time limit (usually 30 minutes) or if the training set becomes empty.

Conceptually, Cedalion is similar to Hydra (Xu, Hoos, and Leyton-Brown 2010) in that both use an algorithm configuration procedure (Hutter 2009) to add the most improving configuration to the existing portfolio in each iteration. However, Hydra uses the algorithm selection system SATzilla (Xu et al. 2008) to select a configuration based on the characteristics of a given test instance, and therefore does not have a notion of time slices. In contrast, Cedalion runs all found configurations sequentially regardless of the instance at hand and makes the length of the time slices part of the configuration space.

Cedalion is also very similar to the greedy algorithm presented by Streeter, Golovin, and Smith (2007). Given a finite set of solvers and their runtimes on the training set, that algorithm iteratively adds the (solver, time slice) pair that most improves the current portfolio per time spent. In contrast, Cedalion does not rely on a priori runtime data and supports infinite sets of solver configurations by using an algorithm configuration procedure to adaptively gather this data for promising configurations only.

In principle, Cedalion could employ any algorithm configuration procedure to select the next (configuration, time slice) pair. Here, we use the model-based configurator SMAC (Hutter, Hoos, and Leyton-Brown 2011) for this task. As a simple standard parallelization method (Hutter, Hoos, and Leyton-Brown 2012), we performed 10 SMAC runs in parallel in every iteration of Cedalion and used the best of the 10 identified (configuration, time slice) pairs.

We could have included planners other than Fast Downward in our Cedalion portfolios (even other parameterized planning frameworks, by configuring on the union of all parameter spaces). This would have almost certainly improved performance, due to the fact that portfolios can exploit the complementary strengths of diverse approaches. Nevertheless, we chose to limit ourselves to Fast Downward in order to quantify the performance gain possible within this framework.

Original Optimization Function

Formally, Cedalion uses the following metric m_P to evaluate $\langle$configuration, time slice$\rangle$ pairs $\langle \boldsymbol{\theta}, t\rangle$ on task π given the current Portfolio P:

$$m_P(\langle \boldsymbol{\theta}, t\rangle, \pi) = \frac{q(P \oplus \langle \boldsymbol{\theta}, t\rangle, \pi) - q(P, \pi)}{t}, \qquad (1)$$

where $P \oplus \langle \boldsymbol{\theta}, t\rangle$ denotes the portfolio P with $\langle \boldsymbol{\theta}, t\rangle$ appended and q is a performance metric. Following IPC evaluation criteria, we define $q(P, \pi)$ as the solution quality achieved by portfolio P for task π, i. e., as the minimum known cost for task π divided by the cost achieved by P. Note that the quality is either 1 or 0 for optimal planning depending on whether P solves π.

Configuration Setup

Our Fast Downward Cedalion portfolios are the result of using Cedalion to find sequential portfolios of Fast Downward (Helmert 2006) configurations. In this section we describe how we used Cedalion to find portfolios for the IPC 2014 sequential satisficing, optimal and agile planning tracks.

Benchmarks

Our set of benchmarks consists of almost all domains from previous IPCs (IPC-1998 – IPC-2011) plus the following additional domains that we included in order to be able to learn on more domains with conditional effects:

- Briefcaseworld from the FF/IPP domain collection (http://fai.cs.uni-saarland.de/hoffmann/ff-domains.html)

- Alarm processing for power networks (Haslum and Grastien 2011)
- Various formulations of the genome edit distance problem (Haslum 2011)
- Synthesis of finite-state controllers (Bonet, Palacios, and Geffner 2009)
- Compilations of conformant planning problems (Palacios and Geffner 2009)

Since the number of tasks per domain varies greatly in this benchmark set, we ran some baseline configurations on the set of instances and only chose the 20 hardest tasks per domain. To this end, for each domain we ignored all unsolved tasks and repeatedly added the task that is solved by the least number of configurations. We broke ties by the runtimes the configurations needed to solve a task. Our hope was that this procedure would yield a benchmark set that focuses training on all domains equally and respects the fact that over time IPC benchmark tasks become more difficult to solve.

Modified Optimization Function

Since Cedalion's optimization function leads to many configurations being added with very small time slices, the number of Cedalion iterations can be quite high, especially for satisficing planning. For the IPC, we therefore adapted the function in a way that makes it focus more on additional quality and less on the time that is needed to achieve it. We achieved this behavior by dividing by $(1 + \log_{10}(t))$ instead of t in Equation 1. In our experiments, this modification led to much fewer iterations while the total quality achieved on the training set did not suffer much.

Satisficing planning

For satisficing planning, the set of Fast Downward configurations Cedalion could choose from was the same as the one used by Fawcett et al. (2011), the only exception being that we also included an implementation of the YAHSP lookahead strategy (Vidal 2004). We used a time budget of 5 hours on 10 machines for every iteration of our portfolio construction process (running 10 parallel independent SMAC runs) and always added the pair of configuration and time slice to the current portfolio that maximized the additional quality score per log time spent.

Optimal planning

For a detailed description of the configuration space for optimal planning, we again refer to Fawcett et al. (2011). In addition to the heuristics mentioned there, we also included incremental LM-cut (Pommerening and Helmert 2013) and the additive CEGAR heuristic (Seipp and Helmert 2014) and allowed the search to reduce partial orders with various instantiations of strong stubborn sets as defined by Wehrle and Helmert (2014).

At the time of the construction of our portfolios the blind and $h^{\max}$ heuristics (Bonet and Geffner 2001) were the only admissible Fast Downward heuristics with conditional effect support. Since planners are required to support this feature in the IPC 2014, we added basic conditional-effect support for the LM-cut (Helmert and Domshlak 2009) and incremental LM-cut and merge-and-shrink (Helmert, Haslum, and Hoffmann 2007) heuristics. In order to also include other heuristics which have no support for conditional effects, we decided to learn two portfolios. The first one was trained on all domains from our benchmark set that do not use conditional effects and Cedalion was allowed to choose from all heuristics. The second one was trained on the domains that require conditional effect support, allowing Cedalion to only choose from the heuristics that support this feature. At runtime our planner checks whether the given task uses conditional effects and selects the appropriate portfolio.

We used 10 SMAC runs of 6 hours each in each Cedalion training iteration and always added the pair of configuration and time slice to the current portfolio that maximizes the number of additional tasks solved per log time spent.

Agile planning

We used the same configuration space and set of benchmarks as for satisficing planning. As mandated by the rules for the IPC 2014 sequential agile planning track we limited the total portfolio time to 300 seconds.

In this setting we used 10 SMAC runs of 10 hours each for each Cedalion training iteration. Due to the evaluation function employed in this track we had to change the optimization function Cedalion uses to find the next pair of configuration and time slice. Instead of preferring the pair that maximizes additional quality per log time spent, we chose the one maximizing the agile score (as defined by the IPC 2014 organizers) per log time spent. Formally, we replace the additional quality q by the additional time quality q_{time}:

$$q_{\text{time}}(P, \pi) = \frac{1}{1 + \log_{10}(t(P, \pi)/t^{\star}(\pi))},$$

where $t(P, \pi)$ is the time portfolio P requires to solve task π (note that $t(P, \pi) = \infty$ if P fails to solve π) and $t^{\star}(\pi)$ is the minimum time any planner needs (approximated by a set of baseline planners). We set $q_{\text{time}}(P, \pi) = 1$ if $t(P, \pi) < t^{\star}(\pi)$ or $t(P, \pi) < 1$.

Acknowledgments

First, we would like to thank all Fast Downward contributors. Portfolios crucially rely on their base algorithms, and as such a strong portfolio is ultimately due to the work of the developers of these base algorithms. We therefore wish to thank Malte Helmert, Jörg Hoffmann, Erez Karpas, Emil Keyder, Raz Nissim, Florian Pommerening, Silvia Richter, Gabriele Röger and Matthias Westphal for their contributions to the Fast Downward codebase.

The portfolios also use code that is as of yet not merged into the main Fast Downward repository. Our thanks go to Yusra Alkhazraji, Malte Helmert, Manuel Heusner, Robert Mattmüller, Manuela Ortlieb, Florian Pommerening, Gabriele Röger and Martin Wehrle for allowing us to include their work in the Cedalion portfolios.

We are also grateful to Patrik Haslum and Héctor Palacios for providing the PDDL tasks.

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 34–41. AAAI Press.

Fawcett, C.; Helmert, M.; Hoos, H.; Karpas, E.; Röger, G.; and Seipp, J. 2011. FD-Autotune: Domain-specific configuration using Fast Downward. In *ICAPS 2011 Workshop on Planning and Learning*, 13–17.

Haslum, P., and Grastien, A. 2011. Diagnosis as planning: Two case studies. In *ICAPS 2011 Scheduling and Planning Applications woRKshop*, 37–44.

Haslum, P. 2011. Computing genome edit distances using domain-independent planning. In *ICAPS 2011 Scheduling and Planning Applications woRKshop*, 45–51.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. AAAI Press.

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

Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2011. Sequential model-based optimization for general algorithm configuration. In Coello, C. A. C., ed., *Proceedings of the Fifth Conference on Learning and Intelligent OptimizatioN (LION 2011)*, 507–523. Springer.

Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2012. Parallel algorithm configuration. In *Proceedings of the Sixth Conference on Learning and Intelligent OptimizatioN (LION 2012)*, 55–70. Springer.

Hutter, F. 2009. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. Ph.D. Dissertation, University of British Columbia.

Palacios, H., and Geffner, H. 2009. Compiling uncertainty away in conformant planning problems with bounded width. *Journal of Artificial Intelligence Research* 35:623–675.

Pommerening, F., and Helmert, M. 2013. Incremental LM-cut. In Borrajo, D.; Kambhampati, S.; Oddi, A.; and Fratini, S., eds., *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*, 162–170. AAAI Press.

Seipp, J., and Helmert, M. 2014. Diverse and additive cartesian abstraction heuristics. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*. AAAI Press.

Seipp, J.; Sievers, S.; and Hutter, F. 2013. Automatic configuration of sequential planning portfolios. Technical Report 5, Universität Basel, Fachbereich Informatik.

Streeter, M. J.; Golovin, D.; and Smith, S. F. 2007. Combining multiple heuristics online. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, 1197–1203. AAAI Press.

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, 150–159. AAAI Press.

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

Xu, L.; Hutter, F.; Hoos, H. H.; and Leyton-Brown, K. 2008. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research* 32:565–606.

Xu, L.; Hoos, H. H.; and Leyton-Brown, K. 2010. Hydra: Automatically configuring algorithms for portfolio-based selection. In Fox, M., and Poole, D., eds., *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2010)*, 210–216. AAAI Press.

Appendix – Fast Downward Cedalion Portfolios

We list the configurations found during the configuration processes of our Fast Downward Cedalion portfolios. Our portfolio components have the form of pairs (time slice, configuration), with the first entry reflecting the time slice allowed for the configuration, which is in turn shown in the second component. Note that if the summed up time slices of the configurations of a portfolio is below the overall time limit, the time slices will be stretched at runtime to match the allowed maximal time, i. e. every time slice is multiplied with the same factor such that the portfolio runs for exactly the overall time limit.

Satisficing Planning

```
1,
--heuristic hGoalCount=goalcount(cost_type=2)
--heuristic hFF=ff(cost_type=0)
--search lazy(alt([single(sum([g(),weight(hFF,10)])),
                   single(sum([g(),weight(hFF,10)]),pref_only=true),
                   single(sum([g(),weight(hGoalCount,10)])),
                   single(sum([g(),weight(hGoalCount,10)]),pref_only=true)],
                  boost=2000),
              preferred=[hFF,hGoalCount],reopen_closed=false,cost_type=1)

142,
--landmarks lmg=lm_rhw(reasonable_orders=false,only_causal_landmarks=false,
                       disjunctive_landmarks=false,conjunctive_landmarks=true,
                       no_orders=false,lm_cost_type=2,cost_type=1)
--heuristic hLM,hFF=lm_ff_syn(lmg,admissible=false)
--heuristic hBlind=blind()
--search lazy(alt([single(sum([g(),weight(hBlind,2)])),
                   single(sum([g(),weight(hBlind,2)]),pref_only=true),
                   single(sum([g(),weight(hLM,2)])),
                   single(sum([g(),weight(hLM,2)]),pref_only=true),
                   single(sum([g(),weight(hFF,2)])),
                   single(sum([g(),weight(hFF,2)]),pref_only=true)],
                  boost=4419),
              preferred=[hLM],reopen_closed=true,cost_type=1)

60,
--heuristic hFF=ff(cost_type=1)
--search lazy(alt([single(sum([g(),weight(hFF,10)])),
                   single(sum([g(),weight(hFF,10)]),pref_only=true)],
                  boost=2000),
              preferred=[hFF],reopen_closed=false,cost_type=1)

121,
--heuristic hAdd=add(cost_type=2)
--heuristic hFF=ff(cost_type=0)
--search lazy(alt([tiebreaking([sum([weight(g(),4),weight(hFF,5)]),hFF]),
                   tiebreaking([sum([weight(g(),4),weight(hFF,5)]),hFF],
                               pref_only=true),
                   tiebreaking([sum([weight(g(),4),weight(hAdd,5)]),hAdd]),
                   tiebreaking([sum([weight(g(),4),weight(hAdd,5)]),hAdd],
                               pref_only=true)],
                  boost=2537),
              preferred=[hFF,hAdd],reopen_closed=true,cost_type=0)

403,
--heuristic hBlind=blind()
--heuristic hAdd=add(cost_type=0)
--heuristic hCg=cg(cost_type=1)
--heuristic hHMax=hmax()
--search eager(alt([tiebreaking([sum([g(),weight(hBlind,7)]),hBlind]),
                    tiebreaking([sum([g(),weight(hHMax,7)]),hHMax]),
```

```
                    tiebreaking([sum([g(),weight(hAdd,7)]),hAdd]),
                    tiebreaking([sum([g(),weight(hCg,7)]),hCg])],
                   boost=2142),
               preferred=[],reopen_closed=true,pathmax=true,lookahead=false,
               cost_type=0)

1,
--heuristic hCea=cea(cost_type=1)
--heuristic hFF=ff(cost_type=2)
--heuristic hBlind=blind()
--search eager(alt([single(sum([g(),weight(hBlind,10)])),
                    single(sum([g(),weight(hBlind,10)]),pref_only=true),
                    single(sum([g(),weight(hFF,10)])),
                    single(sum([g(),weight(hFF,10)]),pref_only=true),
                    single(sum([g(),weight(hCea,10)])),
                    single(sum([g(),weight(hCea,10)]),pref_only=true)],
                   boost=536),
               preferred=[hFF],reopen_closed=false,pathmax=false,
               lookahead=true,la_greedy=false,la_repair=true,cost_type=0)

563,
--landmarks lmg=lm_zg(reasonable_orders=true,only_causal_landmarks=false,
                      disjunctive_landmarks=true,conjunctive_landmarks=true,
                      no_orders=true,cost_type=1)
--heuristic hHMax=hmax()
--heuristic hLM=lmcount(lmg,admissible=false,pref=true,cost_type=1)
--heuristic hCea=cea(cost_type=2)
--heuristic hFF=ff(cost_type=1)
--heuristic hCg=cg(cost_type=2)
--search lazy(alt([tiebreaking([sum([g(),weight(hFF,10)]),hFF]),
                   tiebreaking([sum([g(),weight(hFF,10)]),hFF],pref_only=true),
                   tiebreaking([sum([g(),weight(hLM,10)]),hLM]),
                   tiebreaking([sum([g(),weight(hLM,10)]),hLM],pref_only=true),
                   tiebreaking([sum([g(),weight(hHMax,10)]),hHMax]),
                   tiebreaking([sum([g(),weight(hHMax,10)]),hHMax],pref_only=true),
                   tiebreaking([sum([g(),weight(hCg,10)]),hCg]),
                   tiebreaking([sum([g(),weight(hCg,10)]),hCg],pref_only=true),
                   tiebreaking([sum([g(),weight(hCea,10)]),hCea]),
                   tiebreaking([sum([g(),weight(hCea,10)]),hCea],pref_only=true)],
                  boost=4817),
              preferred=[hFF,hCg],reopen_closed=false,cost_type=1)

1,
--landmarks lmg=lm_rhw(reasonable_orders=false,only_causal_landmarks=false,
                       disjunctive_landmarks=true,conjunctive_landmarks=true,
                       no_orders=true,cost_type=1)
--heuristic hHMax=hmax()
--heuristic hLM=lmcount(lmg,admissible=false,pref=false,cost_type=1)
--heuristic hAdd=add(cost_type=0)
--search lazy(alt([tiebreaking([sum([weight(g(),2),weight(hLM,3)]),hLM]),
                   tiebreaking([sum([weight(g(),2),weight(hHMax,3)]),hHMax]),
                   tiebreaking([sum([weight(g(),2),weight(hAdd,3)]),hAdd])],
                  boost=3002),
              preferred=[],reopen_closed=true,cost_type=0)

62,
--landmarks lmg=lm_zg(reasonable_orders=false,only_causal_landmarks=false,
                      disjunctive_landmarks=true,conjunctive_landmarks=true,
                      no_orders=true,cost_type=0)
```

```
--heuristic hLM=lmcount(lmg,admissible=true,pref=false,cost_type=0)
--search eager(single(sum([g(),weight(hLM,3)])),preferred=[],
               reopen_closed=true,pathmax=false,lookahead=false,cost_type=1)

50,
--landmarks lmg=lm_rhw(reasonable_orders=true,only_causal_landmarks=true,
                       disjunctive_landmarks=true,conjunctive_landmarks=true,
                       no_orders=false,cost_type=2)
--heuristic hBlind=blind()
--heuristic hAdd=add(cost_type=0)
--heuristic hLM=lmcount(lmg,admissible=false,pref=true,cost_type=2)
--heuristic hFF=ff(cost_type=0)
--search lazy(alt([single(sum([weight(g(),2),weight(hBlind,3)])),
                   single(sum([weight(g(),2),weight(hBlind,3)]),pref_only=true),
                   single(sum([weight(g(),2),weight(hFF,3)])),
                   single(sum([weight(g(),2),weight(hFF,3)]),pref_only=true),
                   single(sum([weight(g(),2),weight(hLM,3)])),
                   single(sum([weight(g(),2),weight(hLM,3)]),pref_only=true),
                   single(sum([weight(g(),2),weight(hAdd,3)])),
                   single(sum([weight(g(),2),weight(hAdd,3)]),pref_only=true)],
                  boost=2474),
              preferred=[hAdd],reopen_closed=false,cost_type=1)

188,
--heuristic hCg=cg(cost_type=1)
--heuristic hHMax=hmax()
--heuristic hBlind=blind()
--heuristic hGoalCount=goalcount(cost_type=1)
--search eager(alt([tiebreaking([sum([weight(g(),4),weight(hBlind,5)]),hBlind]),
                    tiebreaking([sum([weight(g(),4),weight(hHMax,5)]),hHMax]),
                    tiebreaking([sum([weight(g(),4),weight(hCg,5)]),hCg]),
                    tiebreaking([sum([weight(g(),4),weight(hGoalCount,5)]),
                                 hGoalCount])],
                   boost=1284),
               preferred=[],reopen_closed=false,pathmax=true,lookahead=false,
               cost_type=1)

2,
--landmarks lmg=lm_exhaust(reasonable_orders=false,only_causal_landmarks=false,
                           disjunctive_landmarks=true,conjunctive_landmarks=true,
                           no_orders=false,lm_cost_type=1,cost_type=0)
--heuristic hGoalCount=goalcount(cost_type=2)
--heuristic hLM,hFF=lm_ff_syn(lmg,admissible=false)
--heuristic hBlind=blind()
--search eager(alt([tiebreaking([sum([weight(g(),8),weight(hBlind,9)]),hBlind]),
                    tiebreaking([sum([weight(g(),8),weight(hLM,9)]),hLM]),
                    tiebreaking([sum([weight(g(),8),weight(hFF,9)]),hFF]),
                    tiebreaking([sum([weight(g(),8),weight(hGoalCount,9)]),
                                 hGoalCount])],
                   boost=2005),
               preferred=[],reopen_closed=true,pathmax=true,lookahead=false,
               cost_type=0)

78,
--heuristic hBlind=blind()
--heuristic hFF=ff(cost_type=1)
--search eager(alt([single(sum([g(),weight(hBlind,2)])),
                    single(sum([g(),weight(hFF,2)]))],boost=4480),
                   preferred=[],reopen_closed=true,pathmax=true,
```

```
                    lookahead=true,la_greedy=true,la_repair=true,cost_type=0)

60,
--heuristic hCea=cea(cost_type=1)
--heuristic hFF=ff(cost_type=2)
--heuristic hGoalCount=goalcount(cost_type=2)
--heuristic hBlind=blind()
--search lazy(alt([tiebreaking([sum([g(),weight(hBlind,10)]),hBlind]),
                   tiebreaking([sum([g(),weight(hBlind,10)]),hBlind],pref_only=true),
                   tiebreaking([sum([g(),weight(hFF,10)]),hFF]),
                   tiebreaking([sum([g(),weight(hFF,10)]),hFF],pref_only=true),
                   tiebreaking([sum([g(),weight(hCea,10)]),hCea]),
                   tiebreaking([sum([g(),weight(hCea,10)]),hCea],pref_only=true),
                   tiebreaking([sum([g(),weight(hGoalCount,10)]),hGoalCount]),
                   tiebreaking([sum([g(),weight(hGoalCount,10)]),hGoalCount],
                               pref_only=true)],
                  boost=2222),
              preferred=[hCea,hGoalCount],reopen_closed=false,cost_type=1)

3,
--heuristic hFF=ff(cost_type=2)
--search lazy(alt([tiebreaking([sum([g(),hFF]),hFF]),
                   tiebreaking([sum([g(),hFF]),hFF],pref_only=true)],
                  boost=432),
              preferred=[hFF],reopen_closed=true,cost_type=1)

50,
--heuristic hGoalCount=goalcount(cost_type=1)
--heuristic hFF=ff(cost_type=2)
--heuristic hBlind=blind()
--heuristic hCg=cg(cost_type=0)
--search lazy(alt([single(sum([weight(g(),2),weight(hBlind,3)])),
                   single(sum([weight(g(),2),weight(hBlind,3)]),pref_only=true),
                   single(sum([weight(g(),2),weight(hFF,3)])),
                   single(sum([weight(g(),2),weight(hFF,3)]),pref_only=true),
                   single(sum([weight(g(),2),weight(hCg,3)])),
                   single(sum([weight(g(),2),weight(hCg,3)]),pref_only=true),
                   single(sum([weight(g(),2),weight(hGoalCount,3)])),
                   single(sum([weight(g(),2),weight(hGoalCount,3)]),pref_only=true)],
                  boost=3662),
              preferred=[hFF],reopen_closed=true,cost_type=0)

14,
--landmarks lmg=lm_zg(reasonable_orders=true,only_causal_landmarks=false,
                      disjunctive_landmarks=true,conjunctive_landmarks=true,
                      no_orders=false,cost_type=1)
--heuristic hCg=cg(cost_type=1)
--heuristic hGoalCount=goalcount(cost_type=0)
--heuristic hHMax=hmax()
--heuristic hCea=cea(cost_type=0)
--heuristic hLM=lmcount(lmg,admissible=false,pref=true,cost_type=1)
--search lazy(alt([single(sum([weight(g(),2),weight(hLM,3)])),
                   single(sum([weight(g(),2),weight(hLM,3)]),pref_only=true),
                   single(sum([weight(g(),2),weight(hHMax,3)])),
                   single(sum([weight(g(),2),weight(hHMax,3)]),pref_only=true),
                   single(sum([weight(g(),2),weight(hCg,3)])),
                   single(sum([weight(g(),2),weight(hCg,3)]),pref_only=true),
                   single(sum([weight(g(),2),weight(hCea,3)])),
                   single(sum([weight(g(),2),weight(hCea,3)]),pref_only=true),
```

```
                    single(sum([weight(g(),2),weight(hGoalCount,3)])),
                    single(sum([weight(g(),2),weight(hGoalCount,3)]),pref_only=true)],
                   boost=2508),
              preferred=[hCea,hGoalCount],reopen_closed=false,cost_type=0)

1,
--landmarks lmg=lm_exhaust(reasonable_orders=false,only_causal_landmarks=false,
                           disjunctive_landmarks=true,conjunctive_landmarks=true,
                           no_orders=false,cost_type=1)
--heuristic hFF=ff(cost_type=2)
--heuristic hHMax=hmax()
--heuristic hBlind=blind()
--heuristic hLM=lmcount(lmg,admissible=true,pref=false,cost_type=1)
--search lazy(alt([single(sum([g(),weight(hBlind,3)])),
                    single(sum([g(),weight(hBlind,3)]),pref_only=true),
                    single(sum([g(),weight(hFF,3)])),
                    single(sum([g(),weight(hFF,3)]),pref_only=true),
                    single(sum([g(),weight(hLM,3)])),
                    single(sum([g(),weight(hLM,3)]),pref_only=true),
                    single(sum([g(),weight(hHMax,3)])),
                    single(sum([g(),weight(hHMax,3)]),pref_only=true)],
                   boost=3052),
              referred=[hFF],reopen_closed=true,cost_type=0)
```

Optimal Planning

- Configurations supporting conditional effects:

```
1,
--heuristic hBlind=blind()
--heuristic hcond_eff_incremental_lmcut=
    cond_eff_incremental_lmcut(local=false,memory_limit=46,
                               keep_frontier=false,reevaluate_parent=false,
                               min_cache_hits=4385)
--heuristic hCombinedMax=max([hBlind,hcond_eff_incremental_lmcut])
--search astar(hCombinedMax,mpd=false,pathmax=true,cost_type=0)

99
--heuristic hMas=
    merge_and_shrink(reduce_labels=true,
                     merge_strategy=MERGE_LINEAR_GOAL_CG_LEVEL,
                     shrink_strategy=
                         shrink_bisimulation(max_states=731,
                                             max_states_before_merge=-1,
                                             greedy=true,
                                             threshold=329,
                                             group_by_h=true,
                                             at_limit=RETURN))
--heuristic hHMax=hmax()
--heuristic hCombinedMax=max([hMas,hHMax])
--search astar(hCombinedMax,mpd=false,pathmax=false,cost_type=0)

38,
--heuristic hcond_eff_incremental_lmcut=
    cond_eff_incremental_lmcut(local=false,memory_limit=94,
                               keep_frontier=false,reevaluate_parent=false,
                               min_cache_hits=9175)
--heuristic hHMax=hmax()
--heuristic hCombinedMax=max([hcond_eff_incremental_lmcut,hHMax])
--search astar(hCombinedMax,mpd=false,pathmax=false,cost_type=0)
```

```
463,
--heuristic hcond_eff_incremental_lmcut=
    cond_eff_incremental_lmcut(local=false,memory_limit=36,
                               keep_frontier=true,reevaluate_parent=false,
                               min_cache_hits=9943)
--search astar(hcond_eff_incremental_lmcut,mpd=false,pathmax=true,cost_type=0)
```

- Configurations not supporting conditional effects:

```
1,
--heuristic hincremental_lmcut=
    incremental_lmcut(local=false,memory_limit=1000,
                      keep_frontier=false,reevaluate_parent=true,
                      min_cache_hits=4294967295)
--heuristic hcpdbs=cpdbs()
--heuristic hCombinedMax=max([hcpdbs,hincremental_lmcut])
--search astar(hCombinedMax,
               partial_order_reduction=
                   sss_expansion_core(active_ops=false,mutexes=none),
               mpd=false,
               pathmax=false,
               cost_type=0)

538,
--landmarks lmg=lm_zg(only_causal_landmarks=false,disjunctive_landmarks=true,
                      conjunctive_landmarks=true,no_orders=true)
--heuristic hincremental_lmcut=
    incremental_lmcut(local=false,memory_limit=200,keep_frontier=false,
                      reevaluate_parent=true,min_cache_hits=4294967295)
--heuristic hLMCut=lmcut()
--heuristic hipdb=ipdb(pdb_max_size=87443270,collection_max_size=182173479,
                       num_samples=94,min_improvement=7,max_time=48)
--heuristic hLM=lmcount(lmg,admissible=true)
--heuristic hCombinedMax=max([hipdb,hLM,hLMCut,hincremental_lmcut])
--search astar(hCombinedMax,mpd=false,pathmax=true,cost_type=0)

338,
--heuristic hHMax=hmax()
--heuristic hCegar=cegar(max_states=6072054,max_time=74,
                         pick=max_constrained,fact_order=original,
                         decomposition=goal_leaves,max_abstractions=5143)
--heuristic hpdb=pdb(max_states=88665)
--heuristic hincremental_lmcut=
    incremental_lmcut(local=false,memory_limit=100,keep_frontier=true,
                      reevaluate_parent=true,min_cache_hits=4294967295)
--heuristic hLMCut=lmcut()
--heuristic hCombinedMax=max([hCegar,hpdb,hLMCut,hincremental_lmcut,hHMax])
--search astar(hCombinedMax,
               partial_order_reduction=
                   small_stubborn_sets(active_ops=false,
                                       precond_choice=minimize_global_var_ordering,
                                       mutexes=fd),mpd=false,
                                       pathmax=true,cost_type=0)

340,
--heuristic hBlind=blind()
--heuristic hpdb=pdb(max_states=61783637)
--heuristic hCegar=cegar(max_states=1052487,max_time=289,
                         pick=max_constrained,fact_order=hadd_down,
```

```
                         decomposition=goal_leaves,max_abstractions=6563)
--heuristic hCombinedMax=max([hCegar,hpdb,hBlind])
--search astar(hCombinedMax,mpd=false,pathmax=false,cost_type=0)

392,
--landmarks lmg=lm_rhw(only_causal_landmarks=false,disjunctive_landmarks=true,
                       conjunctive_landmarks=true,no_orders=false)
--heuristic hMas=
    merge_and_shrink(reduce_labels=true,
                     merge_strategy=MERGE_LINEAR_REVERSE_LEVEL,
                     shrink_strategy=
                         shrink_fh(max_states=10188,max_states_before_merge=-1,
                                   shrink_f=HIGH,shrink_h=HIGH))
--heuristic hpdb=pdb(max_states=958787)
--heuristic hincremental_lmcut=
    incremental_lmcut(local=false,memory_limit=2000,keep_frontier=false,
                      reevaluate_parent=true,min_cache_hits=100)
--heuristic hCegar=cegar(max_states=2970840,max_time=361,pick=max_hadd,
                         fact_order=original,decomposition=landmarks,
                         max_abstractions=473)
--heuristic hcpdbs=cpdbs()
--heuristic hLM=lmcount(lmg,admissible=true)
--heuristic hzopdbs=zopdbs()
--heuristic hCombinedMax=max([hMas,hCegar,hzopdbs,hcpdbs,hpdb,
                              hLM,hincremental_lmcut])
--search astar(hCombinedMax,
               partial_order_reduction=sss_expansion_core(active_ops=true,mutexes=fd),
               mpd=false,pathmax=false,cost_type=0)

113,
--heuristic hcpdbs=cpdbs()
--heuristic hpdb=pdb(max_states=282621)
--heuristic hMas=
    merge_and_shrink(reduce_labels=true,
                     merge_strategy=MERGE_LINEAR_REVERSE_LEVEL,
                     shrink_strategy=
                         shrink_bisimulation(max_states=6447701,
                                             max_states_before_merge=-1,
                                             greedy=false,threshold=4577868,
                                             group_by_h=false,at_limit=RETURN))
--heuristic hLMCut=lmcut()
--heuristic hincremental_lmcut=incremental_lmcut(local=true)
--heuristic hBlind=blind()
--heuristic hipdb=ipdb(pdb_max_size=494641,collection_max_size=2355433,
                       num_samples=1135,min_improvement=11,max_time=21)
--heuristic hCombinedMax=max([hMas,hipdb,hcpdbs,hpdb,hBlind,hLMCut,
                              hincremental_lmcut])
--search astar(hCombinedMax,
               partial_order_reduction=
                   small_stubborn_sets(active_ops=true,
                                       precond_choice=forward,
                                       mutexes=fd),
               mpd=false,pathmax=true,cost_type=0)

11,
--landmarks lmg=lm_rhw(only_causal_landmarks=false,disjunctive_landmarks=true,
                       conjunctive_landmarks=true,no_orders=false)
--heuristic hLM=lmcount(lmg,admissible=true)
--heuristic hLMCut=lmcut()
```

```
--heuristic hcpdbs=cpdbs()
--heuristic hCegar=cegar(max_states=8603232,max_time=6,pick=max_refined,
                         fact_order=hadd_up,decomposition=none,max_abstractions=4284)
--heuristic hHm=hm(m=1)
--heuristic hBlind=blind()
--heuristic hMas=
    merge_and_shrink(reduce_labels=true,
                     merge_strategy=MERGE_LINEAR_GOAL_CG_LEVEL,
                     shrink_strategy=
                         shrink_bisimulation(max_states=3766,
                                             max_states_before_merge=-1,
                                             greedy=false,threshold=2900,
                                             group_by_h=true,at_limit=RETURN))
--heuristic hCombinedMax=max([hMas,hCegar,hcpdbs,hBlind,hHm,hLM,hLMCut])
--search astar(hCombinedMax,mpd=false,pathmax=true,cost_type=0)
```

Agile Planning

```
1,
--heuristic hFF=ff(cost_type=1)
--search eager(alt([single(sum([g(),weight(hFF,10)])),
                    single(sum([g(),weight(hFF,10)]),pref_only=true)],boost=3025),
               preferred=[hFF],reopen_closed=true,pathmax=false,lookahead=true,
               la_greedy=true,la_repair=false,cost_type=1)

49,
--landmarks lmg=lm_rhw(reasonable_orders=true,only_causal_landmarks=false,
                       disjunctive_landmarks=true,conjunctive_landmarks=true,
                       no_orders=false,lm_cost_type=2,cost_type=1)
--heuristic hLM,hFF=lm_ff_syn(lmg,admissible=false)
--search lazy(alt([single(sum([g(),weight(hLM,10)])),
                   single(sum([g(),weight(hLM,10)]),pref_only=true),
                   single(sum([g(),weight(hFF,10)])),
                   single(sum([g(),weight(hFF,10)]),pref_only=true)],boost=2000),
              preferred=[hLM,hFF],reopen_closed=false,cost_type=1)

92,
--heuristic hFF=ff(cost_type=1)
--heuristic hCg=cg(cost_type=1)
--heuristic hBlind=blind()
--search eager(alt([single(sum([g(),weight(hBlind,2)])),
                    single(sum([g(),weight(hFF,2)])),
                    single(sum([g(),weight(hCg,2)]))],boost=4480),
               preferred=[],reopen_closed=true,pathmax=true,lookahead=true,
               la_greedy=true,la_repair=true,cost_type=0)

60,
--landmarks lmg=lm_merged([lm_rhw(),lm_hm(m=1)],reasonable_orders=false,
                          only_causal_landmarks=false,disjunctive_landmarks=true,
                          conjunctive_landmarks=true,no_orders=true,cost_type=2)
--heuristic hLM=lmcount(lmg,admissible=false,pref=false,cost_type=2)
--search lazy(single(hLM),preferred=[],reopen_closed=false,cost_type=0)

60,
--heuristic hAdd=add(cost_type=2)
--search lazy(alt([single(sum([g(),weight(hAdd,10)])),
                   single(sum([g(),weight(hAdd,10)]),pref_only=true)],boost=2000),
              preferred=[hAdd],reopen_closed=false,cost_type=1)
```

Fast Downward Stone Soup 2014

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

Abstract

Fast Downward Stone Soup is a sequential portfolio planner that uses various heuristics and search algorithms that have been implemented in the Fast Downward planning system.

We present the variant participating in the sequential satisficing track of IPC 2014.

Introduction

Fast Downward Stone Soup (Helmert, Röger, and Karpas 2011) is a portfolio planner, based on the Fast Downward planning system (Helmert 2006; 2009), and has first participated in the International Planning Competition in 2011.

In this paper we present the variant for the sequential satisficing track of IPC 2014. It is built on slightly different components than the 2011 variant but uses the same selection method for building the portfolio. Therefore we only briefly recapitulate this procedure and present the resulting portfolio. For a discussion of the algorithm we refer the reader to the planner description paper of Fast Downward Stone Soup 2011 (Helmert, Röger, and Karpas 2011).

Building the Portfolio

We used the same hill-climbing algorithm for building the portfolio as Fast Downward Stone Soup 2011. It requires the following information as input:

- A set of *planning algorithms* $\mathcal{A}$. We used a set of 65 Fast Downward configurations, which we will describe in the next section.
- A set of *training instances* $\mathcal{I}$, for which portfolio performance is optimized. We used a set of 3533 instances, described in the next section.
- Complete *evaluation results* that include, for each algorithm $A \in \mathcal{A}$ and training instance $I \in \mathcal{I}$,
  - the *runtime* $t(A, I)$ of the given algorithm on the given training instance on our evaluation machines, in seconds (we did not consider anytime planners), and
  - the *plan cost* $c(A, I)$ of the plan that was found.

We used a timeout of 30 minutes and memory limit of 2 GB to generate this data. In cases where an instance could not be solved within these bounds, we set $t(A, I) = c(A, I) = \infty$.

build-portfolio(*algorithms*, *results*, *granularity*, *timeout*):
  portfolio := $\{ A \mapsto 0 \mid A \in$ *algorithms* $\}$
  repeat $\lfloor$*timeout*/*granularity*$\rfloor$ **times**:
    candidates := successors(*portfolio*, *granularity*)
    portfolio := $\arg\max_{C \in \textit{candidates}}$ score(C, *results*)
  portfolio := reduce(*portfolio*, *results*)
  return *portfolio*

Figure 1: Algorithm for building a portfolio.

The procedure computes a portfolio as a mapping $P : \mathcal{A} \rightarrow \mathbb{N}_0$ which assigns a time limit (possibly 0 if the algorithm is not used) to each component algorithm. It is a simple hill-climbing search in the space of portfolios, shown in Figure 1.

In addition to the algorithms and the evaluation results, it takes two parameters, *granularity* and *timeout*, both measured in seconds. The timeout is an upper bound on the total time for the generated portfolio, which is the sum of all component time limits. The granularity specifies the step size with which we add time slices to the current portfolio.

The search starts from a portfolio that assigns a time of 0 to all algorithms. In each hill-climbing step, it generates all possible *successors* of the current portfolio. There is one successor per algorithm, where the only difference between the current portfolio and the successor is that the time limit of this algorithm is increased by the given granularity value.

To evaluate the quality of a portfolio, we compute a score in the range 0–1 for each training instance and sum this quantity over all training instances to form a portfolio score.

For each instance, we apply a similar scoring function as used for the International Planning Competitions since 2008, with the only difference that we use the best solution qual-

ity among our algorithms as reference quality: if no algorithm in a portfolio P solves an instance I within its allotted runtime, the instance score is 0. Otherwise, the portfolio is assigned the instance score c_I^*/c_I^P, where c_I^* is the best solution cost for I of any input algorithm $A \in \mathcal{A}$ and c_I^P denotes the best solution cost among all algorithms $A \in \mathcal{A}$ that solve the instance within their allotted runtime $P(A)$.

In each hill-climbing step the search chooses the successor with the highest portfolio score. Ties are broken in favor of successors that increase the timeout of the component algorithm that occurs earliest in some arbitrary total order.

The hill-climbing phase ends when all successors would exceed the given time bound. A post-processing step reduces the time assigned to each algorithm by the portfolio. It considers the algorithms in the same arbitrary order used for breaking ties in the hill-climbing phase and sets their time limit to the lowest number that would still lead to the same portfolio score.

Resulting Portfolio

Our set of training instances consists of almost all tasks from the deterministic track of IPC 1998 – IPC 2011 plus tasks from various other sources: compilations of conformant planning tasks (Palacios and Geffner 2009), finite-state controller synthesis problems (Bonet, Palacios, and Geffner 2009), genome edit distance problems (Haslum 2011), alarm processing tasks for power networks (Haslum and Grastien 2011), and briefcaseworld tasks from the FF/IPP domain collection (http://fai.cs.uni-saarland.de/hoffmann/ff-domains.html). In total, we used 3533 training instances.

For the input planning algorithms, we used the following components:

- *search algorithm:* As in the 2011 variant, we only experimented with greedy best-first search and with weighted A* with a weight of 3.
- *eager vs. lazy:* We again considered both "eager" (textbook) and "lazy" (deferred evaluation) variants of both search algorithms. The work by Richter and Helmert (2009) indicates that both evaluation strategies can be helpful in a portfolio because they have somewhat different strengths and weaknesses.
- *preferred operators:* We used preferred operator information from the heuristics with the default settings of the search algorithms in Fast Downward. For eager search, this is the "dual-queue" method of exploiting preferred operators, for lazy search it is the "boosted dual-queue" method, using a boost value of 1000. This is backed by the results of Richter and Helmert (2009).
- *heuristics:* We used all heuristics used in 2011, which are the additive heuristic h^{add} (Bonet and Geffner 2001), the FF/additive heuristic h^{FF} (Hoffmann and Nebel 2001; Keyder and Geffner 2008), the causal graph heuristic h^{CG} (Helmert 2004), and the context-enhanced additive heuristic h^{cea} (Helmert and Geffner 2008). In addition, we this year included the landmark heuristic h^{LM} (Richter and Westphal 2010) which is known for very good performance when used in combination with h^{FF} as in the LAMA planner (Richter and Westphal 2010) .

  Röger and Helmert (2010) have shown that combinations of multiple heuristics with the "alternation" method can often be very beneficial. Therefore, we considered planner configurations for each of the 10 possible combinations of two of the five heuristics. We did not use larger subsets because computation time for the evaluation results was limited. We also included all single-heuristic configurations except h^{LM} (due to technical problems).

In total, we used 56 planner configurations as input of the hill-climbing procedure. We tried different values for the granularity parameter and achieved the best results (computed from the training set) with a granularity of 40. The resulting portfolio is shown in Tables 1 and 2. It uses 27 of the 56 possible configurations, running them between 17 and 187 seconds. On the training set, the portfolio achieves an overall score of 3234.53, which is much better than the best component algorithm with a score of 2722.17. If we had an oracle to select the perfect algorithm (getting allotted the full 1800 seconds) for each instance, we could reach a total score of 3417.

Sequential Portfolio

In the previous sections, a portfolio simply assigns a runtime to each algorithm, leaving their sequential order open. With the simplifying assumption that all planner runs use the full assigned time and do not communicate information, the order is indeed irrelevant.

In reality, the situation is more complex. First, the Fast Downward planner uses a preprocessing phase that we need to run once before we start the portfolio, so we do not have the full 1800 seconds available. Second, we would like to use the quality of a plan found by one algorithm to prune the search of subsequent planner runs. Third, planner runs often terminate early, e. g. because they run out of memory or find a plan. We would like to use the remaining time to further search for a plan or improve the solution quality. To handle these issues, we employ the same strategy as Fast Downward Stone Soup 2011 in version 1:

We sorted the algorithms by decreasing order of coverage, hence beginning with algorithms likely to *succeed* quickly.

Per-algorithm time limits defined by the portfolio are treated as *relative*, rather than absolute numbers: whenever we start a configuration, we compute the total allotted time of this and all following runs and scale it to the actually remaining computation time. We then assign the respective scaled time to the run. As a result, the last run gets assigned all the remaining time.

The best solution found so far is always used for pruning based on g values: only paths in the state space that are cheaper than the best solution found so far are pursued.

A search algorithm often solves an instance more quickly if it ignores action costs (Richter and Westphal 2010). Therefore we do not take action costs into account until we find the first solution. Afterwards, we re-run the successful configuration using action costs the same way as in the

Search	Evaluation	Heuristics	Performance	Time	Marg. Contribution
Greedy best-first	Eager	$h^{\mathrm{FF}},h^{\mathrm{LM}}$	2722.17 / 3110	35	2.75 / 1
Weighted A*	Eager	$h^{\mathrm{FF}},h^{\mathrm{LM}}$	2663.04 / 2911	40	1.84 / 0
Greedy best-first	Eager	$h^{\mathrm{CG}},h^{\mathrm{FF}}$	2626.41 / 3044	40	2.46 / 0
Weighted A*	Lazy	$h^{\mathrm{FF}},h^{\mathrm{LM}}$	2615.15 / 2962	159	11.71 / 1
Weighted A*	Eager	$h^{\mathrm{CG}},h^{\mathrm{FF}}$	2609.36 / 2927	0	—
Greedy best-first	Eager	$h^{\mathrm{add}},h^{\mathrm{LM}}$	2591.42 / 3116	33	1.45 / 0
Greedy best-first	Lazy	$h^{\mathrm{FF}},h^{\mathrm{LM}}$	2587.26 / 3195	187	19.11 / 19
Greedy best-first	Eager	$h^{\mathrm{add}},h^{\mathrm{FF}}$	2575.28 / 3047	120	12.80 / 12
Greedy best-first	Eager	h^{FF}	2544.79 / 2934	0	—
Weighted A*	Eager	$h^{\mathrm{cea}},h^{\mathrm{FF}}$	2539.53 / 2880	40	2.22 / 1
Weighted A*	Eager	$h^{\mathrm{add}},h^{\mathrm{LM}}$	2539.35 / 2950	0	—
Greedy best-first	Eager	$h^{\mathrm{cea}},h^{\mathrm{FF}}$	2537.93 / 2957	0	—
Weighted A*	Eager	h^{FF}	2535.22 / 2818	72	9.09 / 2
Weighted A*	Eager	$h^{\mathrm{add}},h^{\mathrm{FF}}$	2533.38 / 2904	0	—
Weighted A*	Lazy	$h^{\mathrm{add}},h^{\mathrm{LM}}$	2525.66 / 2998	79	5.13 / 1
Weighted A*	Lazy	$h^{\mathrm{cea}},h^{\mathrm{FF}}$	2522.98 / 2942	39	1.87 / 0
Weighted A*	Eager	$h^{\mathrm{cea}},h^{\mathrm{LM}}$	2518.46 / 2921	37	0.85 / 0
Greedy best-first	Eager	$h^{\mathrm{cea}},h^{\mathrm{LM}}$	2512.55 / 2982	0	—
Weighted A*	Lazy	$h^{\mathrm{cea}},h^{\mathrm{LM}}$	2511.75 / 2957	39	2.19 / 0
Greedy best-first	Eager	$h^{\mathrm{add}},h^{\mathrm{CG}}$	2510.72 / 3016	0	—
Weighted A*	Lazy	$h^{\mathrm{CG}},h^{\mathrm{FF}}$	2507.10 / 2918	40	2.48 / 0
Weighted A*	Eager	$h^{\mathrm{CG}},h^{\mathrm{LM}}$	2505.67 / 2857	78	3.50 / 0
Greedy best-first	Eager	$h^{\mathrm{CG}},h^{\mathrm{LM}}$	2505.49 / 2955	78	8.20 / 3
Greedy best-first	Lazy	$h^{\mathrm{add}},h^{\mathrm{LM}}$	2492.53 / 3199	114	5.77 / 3
Greedy best-first	Lazy	$h^{\mathrm{cea}},h^{\mathrm{FF}}$	2487.23 / 3035	0	—
Weighted A*	Eager	$h^{\mathrm{add}},h^{\mathrm{CG}}$	2478.44 / 2909	0	—
Greedy best-first	Lazy	$h^{\mathrm{CG}},h^{\mathrm{FF}}$	2470.78 / 3042	0	—
Greedy best-first	Eager	h^{add}	2464.16 / 2994	0	—
Weighted A*	Eager	h^{add}	2446.85 / 2909	77	5.52 / 3
Greedy best-first	Lazy	$h^{\mathrm{cea}},h^{\mathrm{LM}}$	2434.88 / 3070	39	9.00 / 8
Weighted A*	Lazy	$h^{\mathrm{add}},h^{\mathrm{FF}}$	2428.48 / 2940	0	—
Weighted A*	Lazy	$h^{\mathrm{CG}},h^{\mathrm{LM}}$	2415.80 / 2839	39	4.59 / 0
Greedy best-first	Eager	$h^{\mathrm{cea}},h^{\mathrm{CG}}$	2407.77 / 2899	0	—
Weighted A*	Eager	$h^{\mathrm{cea}},h^{\mathrm{CG}}$	2406.41 / 2825	0	—
Weighted A*	Eager	$h^{\mathrm{cea}},h^{\mathrm{add}}$	2403.75 / 2837	0	—
Greedy best-first	Lazy	$h^{\mathrm{CG}},h^{\mathrm{LM}}$	2380.44 / 2980	0	—
Weighted A*	Lazy	h^{FF}	2372.10 / 2801	38	2.86 / 1
Weighted A*	Eager	h^{cea}	2366.58 / 2803	0	—
Greedy best-first	Lazy	$h^{\mathrm{add}},h^{\mathrm{FF}}$	2365.67 / 3031	17	2.17 / 2
Greedy best-first	Lazy	h^{FF}	2350.87 / 2941	0	—
Weighted A*	Lazy	$h^{\mathrm{cea}},h^{\mathrm{CG}}$	2336.09 / 2852	0	—
Greedy best-first	Eager	h^{cea}	2330.01 / 2845	40	2.75 / 2
Greedy best-first	Eager	$h^{\mathrm{cea}},h^{\mathrm{add}}$	2324.89 / 2794	0	—
Weighted A*	Lazy	$h^{\mathrm{add}},h^{\mathrm{CG}}$	2320.28 / 2875	0	—
Greedy best-first	Lazy	$h^{\mathrm{add}},h^{\mathrm{CG}}$	2307.49 / 2999	40	3.47 / 0
Greedy best-first	Eager	h^{CG}	2290.74 / 2713	0	—
Weighted A*	Lazy	$h^{\mathrm{cea}},h^{\mathrm{add}}$	2285.72 / 2830	0	—

Table 1: Fast Downward Stone Soup 2014 (continued in Table 2). The performance column shows the score/coverage of the configuration over all training instances. The last column shows the decrease of score and number of solved instances when removing only this configuration from the portfolio.

Search	Evaluation	Heuristics	Performance	Time	Marg. Contribution
Greedy best-first	Lazy	h^{cea},h^{CG}	2281.19 / 2941	0	—
Weighted A*	Eager	h^{CG}	2271.04 / 2612	38	3.10 / 0
Weighted A*	Lazy	h^{cea}	2269.16 / 2820	40	2.50 / 0
Weighted A*	Lazy	h^{add}	2238.18 / 2852	0	—
Weighted A*	Lazy	h^{CG}	2205.40 / 2631	0	—
Greedy best-first	Lazy	h^{CG}	2200.30 / 2762	116	12.77 / 10
Greedy best-first	Lazy	h^{cea},h^{add}	2189.99 / 2844	0	—
Greedy best-first	Lazy	h^{cea}	2187.15 / 2900	0	—
Greedy best-first	Lazy	h^{add}	2181.16 / 2958	0	—
Portfolio			3234.53 / 3286	1714	
"Holy Grail"			3417.00 / 3417		

Table 2: Fast Downward Stone Soup 2014 (continuation of Table 1).

LAMA planner, by treating all actions of cost c with cost $c + 1$ in the heuristic and using the true action costs in the search component. We maintain this strategy for all remaining planner runs.

Conclusion

Fast Downward Stone Soup 2014 is a very simple portfolio planner. We are aware that our approach is in almost every respect not state of the art in portfolio computation, machine learning, or parameter tuning. Even though, since the 2011 variant was the runner-up at IPC 2011, we decided to submit it nevertheless as a baseline for other portfolio planners in the competition.

Acknowledgments

It is a matter of fact that for a portfolio planner not those who *combined* the components deserve the main credit but those who *contributed* these components.

We therefore wish to thank Blai Bonet, Héctor Geffner, Malte Helmert, Jörg Hoffmann, Emil Keyder, and Silvia Richter, who devised the heuristics used in the portfolio. We could also build on extensive studies by Silvia Richter and Malte Helmert on the influence of different evaluation methods and preferred operators.

Special credit also goes to the core developers of Fast Downward who steadily maintain the code basis with a significant amount of work that often goes unnoticed: Malte Helmert, Erez Karpas, and Silvan Sievers (and – less important – the authors of this paper).

References

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *AIJ* 129(1):5–33.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *Proc. ICAPS 2009*, 34–41.

Haslum, P., and Grastien, A. 2011. Diagnosis as planning: Two case studies. In *ICAPS 2011 Scheduling and Planning Applications woRKshop*, 37–44.

Haslum, P. 2011. Computing genome edit distances using domain-independent planning. In *ICAPS 2011 Scheduling and Planning Applications woRKshop*, 45–51.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In *Proc. ICAPS 2008*, 140–147.

Helmert, M.; Röger, G.; and Karpas, E. 2011. Fast Downward Stone Soup: A baseline for building planner portfolios. In *ICAPS 2011 Workshop on Planning and Learning*, 28–35.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proc. ICAPS 2004*, 161–170.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *AIJ* 173:503–535.

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

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proc. ECAI 2008*, 588–592.

Palacios, H., and Geffner, H. 2009. Compiling uncertainty away in conformant planning problems with bounded width. *JAIR* 35:623–675.

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

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR* 39:127–177.

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In *Proc. ICAPS 2010*, 246–249.

Fast Downward Uniform Portfolio

Jendrik Seipp
Universität Basel
Basel, Switzerland
jendrik.seipp@unibas.ch

Manuel Braun
Johannes Garimort
Albert-Ludwigs-Universität Freiburg
Freiburg, Germany

The Fast Downward uniform portfolio runs 21 automatically configured Fast Downward instantiations sequentially for the same amount of time. The portfolio is identical to the "uniform" portfolio in Seipp et al. (2012a). Therefore, we only give a high-level description here and refer to the paper for details of its construction and an exerimental analysis.

In a nutshell, our uniform portfolio approach works as follows: we used the automatic parameter tuning framework ParamILS (Hutter et al. 2009) to find fast configurations of the Fast Downward planning system for 21 planning domains separately. At runtime we run all found configurations sequentially for the same amount of time, i.e. in the IPC setting with a time limit of 30 minutes, all configurations run for at most 85 seconds.

The details of our approach can be found in Seipp et al. (2012a) and the accompanying technical report (Seipp et al. 2012b).

References

Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.

Seipp, J.; Braun, M.; Garimort, J.; and Helmert, M. 2012a. Learning portfolios of automatically tuned planners. In McCluskey, L.; Williams, B.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS 2012)*. AAAI Press.

Seipp, J.; Braun, M.; Garimort, J.; and Helmert, M. 2012b. Learning portfolios of automatically tuned planners: Detailed results. Technical Report 268, Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

The Freelunch Planning System Entering IPC 2014

Tomáš Balyo
Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics, Charles University in Prague
biotomas@gmail.com

Abstract

Freelunch is an open-source planning system written in Java. It takes input in the multivalued SAS+ format, which can be obtained from PDDL by Helmert's translation tool. Freelunch also provides a Java API to make its use in Java applications convenient. The philosophy of Freelunch is to first find a plan of arbitrary quality and then improve it using post planning optimization techniques. Several algorithms are implemented in Freelunch, many of them are based on translation of planning problems into satisfiability (SAT) and using a SAT solver.

Introduction

Freelunch (http://ktiml.mff.cuni.cz/freelunch) is an open-source Java planner and planning library. It is designed to be used in Java applications, for example to implement sophisticated artificial intelligence for games. Freelunch is a collection of planning algorithms sharing a common interface, which is easy to understand and use. Everything is written in Java, which allows using the library on any device or platform that supports Java.

There are many other planners which are faster than Freelunch, but they are mostly written in C or C++. They are using a formalism called PDDL, which is in our opinion rather hard to use. Describing a planning problem correctly in PDDL requires some skills and experience with abstract modeling. On the other hand describing a problem with Freelunch is easy and straightforward

Freelunch can be used as a command line planner, it accepts the SAS+ format defined by Helmert (Helmert 2014). However, most of the planning benchmark problems come in the PDDL format, which can be translated into SAS files using Helmert's translation tool, which is a part of the Fast Downward planner (Helmert 2006).

This paper is focused on the description of the algorithms used for the 2014 planning competition version of the command line planner. The main algorithm can be summarized as follows. First, we translate the PDDL input into the SAS+ formalism. Then we run a simple backtracking search, which can solve some of the domains very quickly (at the expense of very long plans). If the backtracking fails to find a plan in the given time limit, then we switch to a SAT based approach. If a plan is found, then post-planning optimization techniques are used to increase its quality.

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Preliminary Definitions

In this section we give the formal definitions related to planning. We will use the multivalued SAS+ formalism (Bäckström and Nebel 1995) instead of the classical STRIPS formalism (Fikes and Nilsson 1971) based on propositional logic.

A planning task Π in the SAS+ formalism is defined as a tuple $\Pi = \{X, O, s_I, s_G\}$ where

- $X = \{x_1, \ldots, x_n\}$ is a set of multivalued variables with finite domains $\mathrm{dom}(x_i)$.
- O is a set of actions (or operators). Each action $a \in O$ is a tuple $(\mathrm{pre}(a), \mathrm{eff}(a))$ where $\mathrm{pre}(a)$ is the set of preconditions of a and $\mathrm{eff}(a)$ is the set of effects of a. Both preconditions and effects are of the form $x_i = v$ where $v \in \mathrm{dom}(x_i)$.
- A state is a set of assignments to the state variables. Each state variable has exactly one value assigned from its respective domain. We denote by S the set of all states. $s_I \in S$ is the initial state. s_G is a partial assignment of the state variables (not all variables have assigned values) and a state $s \in S$ is a goal state if $s_G \subseteq s$.

An action a is *applicable* in the given state s if $\mathrm{pre}(a) \subseteq s$. By $s' = \mathrm{apply}(a, s)$ we denote the state after executing the action a in the state s, where a is applicable in s. All the assignments in s' are the same as in s except for the assignments in $\mathrm{eff}(a)$ which replace the corresponding (same variable) assignments in s.

A *(sequential) plan* P of *length* k for a given planning task Π is a sequence of actions $P = \{a_1, \ldots, a_k\}$ such that $s_G \subseteq \mathrm{apply}(a_k, \mathrm{apply}(a_{k-1} \ldots \mathrm{apply}(a_2, \mathrm{apply}(a_1, s_I)) \ldots))$. We will denote by $|P|$ the length of the plan.

Finding Plans

The planner version entering the IPC 2014 can be considered a simple sequential portfolio planner. The portfolio consists of only two planning algorithms. The first is a simple heuristic forward search algorithm, the second is a SAT based approach. The time limit for the forward search is set to 10

seconds, the rest of the time is used for SAT search and post planning optimization. A more detailed description of both algorithms follows.

Heuristic Forward Search

Starting with the initial state, the algorithm computes all the applicable actions in the current state that lead to a not yet visited state. For each of these actions a heuristic value is computed representing its supposed usefulness. The action with the highest value is selected and applied on the current state. If we get to a state, that each applicable action leads to an already visited state or there is no applicable action, then we backtrack to the previous state.

The heuristic function of action usefulness is very simple and greedy. An action starts with a score of 0. If an effect of the action sets a variable to a goal value while in the current state it has a different value, then the score of the action is increased by 1. On the other hand, if an effect changes the value of a variable which already has a goal value, then the score is decreased by 1. Finally, to break the ties, the score is multiplied by 10 and a random value between 0 and 9 is added to it.

Despite its simplicity, this algorithm can solve around one half of the IPC 2011 benchmark problems very quickly. The downside of the algorithm is that it finds extremely long plans full of redundant actions. For example, for domains such as Elevators and Transport the found plans contain hundreds of thousands of actions while plans found by Fast Downward (Helmert 2006) only have a few hundred actions.

Fortunately, these extremely long plans can be easily reduced to reasonable lengths using post planning optimization techniques. Even the simplest such techniques perform very well on these plans due to their severe redundancy.

SAT Based Search

The SAT based algorithm uses the Relaxed Relaxed Exist-Step (RRES) encoding (Balyo 2013). This novel encoding is designed to allow more actions to be put inside one parallel step than other encodings and thus a planning problem can be solved with fewer SAT solver calls. We use the state-of-the-art SAT solver Lingeling (Biere 2013) to solve the formulas.

Post Planning Optimization

Freelunch entered two tracks of the IPC 2014. One is the agile track, where the quality of the plans is not considered, the other is the satisficing track, where both the time of finding a plan and its quality is important. Therefore, the satisficing track version of Freelunch does post planning optimization on the found plan to increase its quality.

The first post planning optimization algorithm run on the plan is Action Elimination (AE) (Nakhost and Müller 2010; Fink and Yang 1992). AE is a polynomial ($O(|P|^2)$) heuristic algorithm capable of removing redundant (unnecessary) actions from plans. It is not guaranteed to remove all redundant actions (which is an NP-complete problem (Fink and Yang 1992)) and it cannot add/replace actions.

The second method used for plan improvement is a local re-planning based algorithm (Balyo, Barták, and Surynek 2012). It is an anytime technique for decreasing the plan length via substituting parts of the plan by optimal sub-plans. The optimal sub-plans are found using a SAT based planner using the SASE encoding (Huang, Chen, and Zhang 2010) and the Sat4j Java SAT solver (Berre and Parrain 2010). The technique guarantees optimality though it is primarily intended to quickly improve plan quality. We run this algorithm until an optimal plan is reached or the planner is killed due to reaching the time limit.

Conclusion

In this paper we described a subset of the algorithms contained in the Freelunch planning library that was selected for the competition version of the planner entering the IPC 2014. We hope, that the forward search and the SAT based algorithm will complete each other to solve many instances of the competition's benchmark problems.

Acknowledgments

The research is supported by the Czech Science Foundation under the contract P103/10/1287 and by the Grant Agency of Charles University under contract no. 600112. This research was also supported by the SVV project number 267 314.

References

Bäckström, C., and Nebel, B. 1995. Complexity results for sas+ planning. *Computational Intelligence* 11:625–656.

Balyo, T.; Barták, R.; and Surynek, P. 2012. Shortening plans by local re-planning. In *Proceedings of ICTAI*, 1022–1028.

Balyo, T. 2013. Relaxing the relaxed exist-step parallel planning semantics. In *ICTAI*, 865–871. IEEE.

Berre, D. L., and Parrain, A. 2010. The sat4j library, release 2.2. *JSAT* 7(2-3):59–6.

Biere, A. 2013. Lingeling and plingeling home page. http://fmv.jku.at/lingeling/.

Fikes, R., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artif. Intell.* 2(3/4):189–208.

Fink, E., and Yang, Q. 1992. Formalizing plan justifications. In *In Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence*, 9–14.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Helmert, M. 2014. Output of the fast downward translator. http://www.fast-downward.org/TranslatorOutputFormat.

Huang, R.; Chen, Y.; and Zhang, W. 2010. A novel transition based encoding scheme for planning as satisfiability. In Fox, M., and Poole, D., eds., *AAAI*. AAAI Press.

Nakhost, H., and Müller, M. 2010. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In Brafman, R. I.; Geffner, H.; Hoffmann, J.; and Kautz, H. A., eds., *ICAPS*, 121–128. AAAI.

IBACOP and IBACOP2 Planner

Isabel Cenamor and **Tomás de la Rosa** and **Fernando Fernández**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. Leganés (Madrid). Spain
icenamor@inf.uc3m.es, trosa@inf.uc3m.es, ffernand@inf.uc3m.es

Abstract

This manuscript describes several planning portfolios that use the same base planners. Our Instance Based Configured Portfolios follow two different strategies. IBaCoP is configured a priori following a Pareto efficiency approach to select a sub-set of planners (baseline strategy), which receive the same execution time for all planning problems. On the contrary, IBaCoP2 decides for each problem the sub-set of planners to use. Such decisions are based on predictive models learnt also with training instances gathered from previous executions of the base planners. Both portfolios compete in the sequential satisficing, agile and multi-core tracks.

Introduction

In the state of the art, there are several portfolios that define different ways to combine simple base planners. All of them are motivated by the general idea that none of existing planners dominates all other in all cases. The most popular strategy is the static, where the portfolio components and the time for each planner is defined "a priori" and maintained for all domains and problems. Fast Downward Stone Soup (FDSS) (Helmert 2006) is an example of this type of portfolio. It has various configuration based on previous planning results. It obtained good results in the last IPC (*International Planning Competition*).

The family of PbP portfolios, PbP and PbP2 (Gerevini, Saetti, and Vallati 2009; 2011), generates domain-specific multi-planners from a set of domain-independent planning techniques. It generates macro-actions, optimizes planner parameters and selects specific planners for each domain. Therefore, it generates a different configuration for each domain. This family of portfolios won both learning tracks that were held in past IPCs.

All our portfolios use as base planners the ones competing in the last IPC in the sequential satisficing track (27 planners), plus LPG-tn. The different configurations are obtained applying two different strategies. One is a static configuration that obtains a sub-group of planners (IBaCoP), and the other is a dynamic portfolio configured through CRISP-Data Mining methodology (Chapman et al. 2000; Han, Kamber, and Pei 2006) (IBaCoP2).

IBaCoP is the result of applying the Pareto efficiency technique (Censor 1977) to select a sub-set of planners from the planners in sequential satisficing track plus LPG-tn. We select the planners that dominates all others in at least one domain (from a set of training domains), taking into account quality and time. We assign the same running time for all selected planners

IBaCoP2 is a portfolio auto configurable with a classification model. This portfolio is an evolution of IBaCoP, since it takes as base planners, the planners selected by IBaCoP. However, IBaCoP2 performs a second planner selection using predictive models. These models are the result of learning processes and predict the behaviour of the planners in future problems, i.e. whether they will be able to solve the problems or not. The planners with higher confidence are selected and ordered following such confidence. Then, running time is divided uniformly among them.

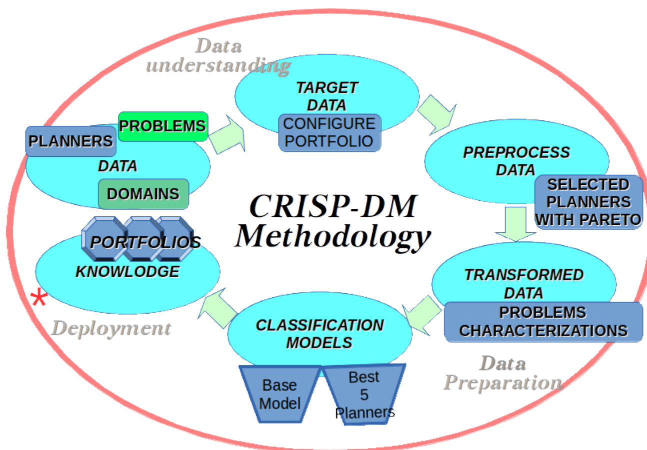
The remainder of the paper is organized as follows. In the next section we present the general ideas of the portfolios, with their components, the training data, and how we finally created the portfolios. We finish with the specific information of the planners in the different tracks.

General System Description

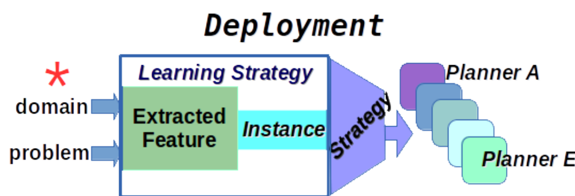
In this section, we explain the general process to configure IBaCoP planners. This system is based on a CRISP-Data Mining and the general idea is depicted in the figure 1.

The first phase in the methodology is to understand the aims of the data mining process: to extract knowledge from past planning executions to create portfolio configurations. Another important consideration is the available inputs of the process. We have two different inputs: the planners, that were extracted from the last International Planning Competition (IPC) plus LPG-tn; and the domains and the problems from different competitions and different tracks. The output of the process is the configuration of the portfolio, which comprises the way to combine the initial planner components and the assigned time per planner.

The next step is to select and preprocess the data. The data is obtained from several sources as explained later, but basically, it can be considered as information about the execution of each planner for each problem of every domain. However, only execution data from the selected planners by the Pareto efficiency technique are used in following steps. In addition, data from problems that were not solved by any planner was eliminated.



(a) General Process



(b) Deployment

Figure 1: General CRISP-DM Process in IBaCoP system

The following step is problem characterization. We have created some features to better differentiate the planning executions. In this phase, we also choose the output attribute in the learning process, which is whether the planner found a solution for the problem in a 1800 seconds.

To continue with the process, we select and apply a variety of modelling techniques to find the model with higher accuracy. In this part of the process, it is necessary the division of the data into training and test sets, which allow us to estimate the future performance of the models. The output models should be evaluated in the context of the business objectives established in the first phase, i.e. planning capability of the developed portfolio.

The last phase, the deployment, is the part of the process that verifies previously held hypotheses through the knowledge discovered in the earlier phases of the CRISP-DM process. Particularly, this deployment appears in Figure 1(b), where the final system gets a new problem and domain, calculate the features, queries to the strategy, and returns the planners with their runtime.

Data understanding

The first step in the DM process is to know the final objective. In our case, it is the configuration of portfolios through the methodology. In the following step, preprocess data, we analyze all the possibilities for the input of the system (domains, problems and planners) and decide the best possible selection. In the case of the planners, we started with all planners from the sequential satisficing track in the last IPC plus LPG-tn. Nevertheless, there are some planners that obtained similar results, and that do not contribute diversity to the portfolio. The chosen planners are selected by using the Pareto efficiency (Censor 1977) technique between the quality of the best solution found and the time (in seconds) of the first solution to be found by the planner. Next, it gives to each planner a score that equals the number of tuples it Pareto-dominates for the same task.

Selected Planners The Pareto efficiency, performed with the results of IPC 2011, outputs 11 planners plus LPG-tn planner:

- ARVAND (Nakhost, Valenzano, and Xie 2011)
- FD-AUTOTUNE 1 Y 2 (Fawcett et al. 2011)
- FD STONE SOUP (FDSS) 1 Y 2 (Helmert et al. 2011)
- LAMA 2008 Y 2011 (Richter, Westphal, and Helmert 2011)
- PROBE (Lipovetzky and Geffner 2011)
- MADAGASCAR (Rintanen 2011)
- RANDWARD (Olsen and Bryce 2011)
- YAHSP2-MT (Vidal 2011)
- LPG-TN (Gerevini et al. 2004)

Selected Domains The next step is to define the set of problems and domains used to learn the models. We need a wide group of problems and domains to generalize properly. We have included the planning problems available from past IPCs, discarding the first four competitions given that problems are too easy for the state-of-the-art planners.

- IPC5: openstack, pathways, rover, storage, tpp and trucks.
- IPC6: cybersec, elevators, openstack, parcprinter, pegsol, pipesworld, scanalyzer, sokoban, transport and woodworking.
- IPC7: barman, elevators, floortile, nomystery, visitall, tidybot, openstacks, parcprinter, parking, pegsol, sokoban, scanalyzer, transport and woodworking
- Leaning 2008: gold-miner, matching-bw, n-puzzle, parking and sokoban.
- Leaning 2011: barman, blockworld, depots, gripper, parking, rover satellite, spanner and tpp.

We consider all the successful problems, and we did not take into account repeated domains or repeated problems. We do not know which domains would be used in the future, so we need to consider a wide and significant number of instances for the learning process. Finally, we obtained 1070 different problems to create the learning models.

Data Preparation

The next step is the characterization of the problem (transformed data in DM process). For this task we consider some features in the planning task previously used (Roberts and Howe 2009) and include others for a better particularization of the problems complexity (Cenamor, de la Rosa, and

Fernández 2012). These features have shown good accuracy for configuring portfolios (Cenamor, de la Rosa, and Fernández 2013). In addition, we create some new features to improve the characterization of the initial state of the problem.

Some basic features are directly extracted from the PDDL files. A group of elaborated features are generated from the problem translation to the SAS$^+$ formalism (Backstrom and Nebel 1995) and its induced graphs, i.e., the causal graph and the domains transition graphs. These features describe number of edges, weights, variables of the graphs. Besides, we include statistical information of the graphs, such as the sum, maximum and standard deviation of the edges and weights. We also consider other information that appears in the translation and preprocess of Fast Downward (Helmert 2006) (FD) system.

As new features we include the most representative heuristic functions computed for the initial state with unit cost, the ratio h_{FF}/h_{max} and a set of features to characterize the *fact balance* of the relaxed plan (RP). We define the fact balance for fact p, as the number of times p appears as an add effect of an action belonging to RP, minus the number of times p is a delete effect of an action in RP, considering original actions where deletes are not ignore. The intuition behind fact balances is that high positive values would characterize easier (relaxed) problems for a given domain, since achieved facts need to be deleted many times. Given that the number of relevant facts of a planning task is variable, we compute statistics (i.e., min, max, average and variance) for the fact balance of the relevant facts. Additionally, we compute statistics only considering facts that are goals, following the same procedure.

The time to extract features is negligible given that features wrt. graphs imply basic arithmetic computations and heuristic functions are only called once for the initial state. To finalize the data preparation, we perform a feature selection process where we get the same performance with a subgroup of all features (35 features). Such features are:

- From the previous work (Cenamor, de la Rosa, and Fernández 2012), we include the number of objects, the number of goals, the number of variables in the causal graph (CG), the ratio between the high level variable and all variables in the CG, the standard deviation of the number of input edges in the CG, the average of the number of output edges in the CG, the maximum and the average weight of the output edges in the CG, the standard deviation of the number of output edges in high level variables in the CG, the maximum weight of input edges in high level variables in the CG, the number of variables in the domain transition graph (DTG), the number of edges in the same graph and the maximum weight of input edges in the DTG.
- As new features from previous work we include: the number of types of objects, the number of functions, the number of auxiliary atoms in the translate process between PDDL to SAS$^+$, the number of implied effect removed, the number of translator facts and the number of the mutex group in the translator process. In addition, the feature selection includes the number of relevant facts, the number of actions, the ratio h_{FF}/h_{max}, the fact balance (average and variance), the goal balance (minimum, average and variance). As well as the following heuristics: h_{add}, h_{max} (Bonet and Geffner 2001), Context enhanced additive (Helmert and Geffner 2008), h_{FF} (Hoffmann and Nebel 2011), Goal count (i.e., the number of unsatisfied goals), Landmark count (Richter, Helmert, and Westphal 2008) and Landmark cut (Helmert and Domshlak 2009) .

Modelling the Data

One of the most important steps in this system is learning classification models to predict whether a planner will find a solution for a problem. We trained with 25 classification algorithms (for different model types: trees, rules, support vector machines and instance based learning) using WEKA (Witten and Frank 2005). WEKA is a data mining toolkit that provides a standard format for running machine learning algorithms. We selected the model with best accuracy (99.83% in training phase): Random Forest (Breiman 2001). This model is a combination of tree predictors such that each tree depends on the values of a random vector and with the same distribution.

In addition, we include two strategies to compare the performance of the system. The first one selects planners only with the Pareto efficiency, and the other uses in addition the classification model.

Deployment

In this section, we explain the different configurations of the system in the different tracks (sequential satisficing, sequential agile and sequential multi-core). The summary is reported in Table 1.

Sequential Satisficing Planner

The IBaCoP planner uses the 12 planners described in subsection Selected Planners, which are selected by the Pareto efficiency analysis. The execution order of the planners is arbitrary, since time is divided uniformly among all them (150 seconds per planner).

The IBaCoP2 planner uses the learned model described in section Modelling the Data . It selects the 5 planners with the highest confidence of solving the problem. The execution order of the planners is based on their confidence. The running time is assigned uniformly to each planner (360 seconds).

Sequential Agile Planner

As in the Sequential Satisficing track, the IBaCoP planner uses the 12 planners described in subsection Selected Planners. However, we assigned as running time the average time to find the first solution in 300 seconds. The execution order for the planners is given by this average from less time to grater values. Even though all planners are included, in practice, only a few of them will have the chance to run, until consuming the time bound of 300 seconds.

The IBaCoP2 planner uses the learned model to select 5 planners; the order of the planners is decided from the confidence and the time for each planner is the same for the five planners.

Sequential Multi-core Planner

For Multi-core track, the planners are the same as for the sequential satisficing track, but taking into account that we have more time ($1800\ seconds \times 4\ cores$). Memory is divided equally among all the planners running in the different cores.

	Planner	seq-sat	seq-agl	seq-mco
1	yahsp2-mt	150	5	600
2	randward	150	50	600
3	arvand	150	55	600
4	fd-autotune-1	150	50	600
5	lama-2008	150	45	600
6	probe	150	–	600
7	madagascar	150	45	600
8	lpg-tn	150	50	600
9	fdss-1	150	–	600
10	lama-2011	150	–	600
11	fd-autotune-2	150	–	600
12	fdss-2	150	–	600

Table 1: Time for each planner execution in sequential tracks

Acknowledgements

We generated sequential portfolios of existing planners to be submitted to the International Planning Competition 2014. Thus, we would like to acknowledge and thank the authors of the individual planners for their contribution and hard work.

This work has been partially supported by the Spanish project TSI-090302-2011-6 and TIN2012-38079-C03-02.

References

Backstrom, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11:625–655.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Breiman, L. 2001. Random forests. *Machine learning* 45(1):5–32.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2012. Mining ipc-2011 results. In *Proceedings of the Third Workshop on the International Planning Competition*.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2013. Learning predictive models to configure planning portfolios. In *Proceedings of the Workshop on the Planning and Learning*.

Censor, Y. 1977. Pareto optimality in multiobjective problems. *Applied Mathematics and Optimization* 4(1):41–59.

Chapman, P.; Clinton, J.; Kerber, R.; Khabaza, T.; Reinartz, T.; Shearer, C.; and Wirth, R. 2000. Crisp-dm 1.0 step-by-step data mining guide. *CRISPWP-0800*.

Fawcett, C.; Helmert, M.; Hoos, H.; Karpas, E.; Röger, G.; and Seipp, J. 2011. Fd-autotune: Automated configuration of fast downward. *The 2011 International Planning Competition* 31–37.

Gerevini, A.; Saetti, A.; Serina, I.; and Toninelli, P. 2004. Lpg-td: a fully automated planner for pddl2. 2 domains. In *In Proc. of the 14th Int. Conference on Automated Planning and Scheduling (ICAPS-04) International Planning Competition abstracts*. Citeseer.

Gerevini, A.; Saetti, A.; and Vallati, M. 2009. An automatically configurable portfolio-based planner with macro-actions: PbP. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-09)*.

Gerevini, A.; Saetti, A.; and Vallati, M. 2011. Pbp2: Automatic configuration of a portfolio-based multi-planner. *The 2011 International Planning Competition*.

Han, J.; Kamber, M.; and Pei, J. 2006. *Data mining: concepts and techniques*. Morgan kaufmann.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In *ICAPS*, 140–147.

Helmert, M.; Röger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S.; and Westphal, M. 2011. Fast downward stone soup. *The 2011 International Planning Competition* 38.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26(1):191–246.

Hoffmann, J., and Nebel, B. 2011. The ff planning system: Fast plan generation through heuristic search. *arXiv preprint arXiv:1106.0675*.

Lipovetzky, N., and Geffner, H. 2011. Searching with probes: The classical planner probe. *The 2011 International Planning Competition* 30(29):71.

Nakhost, H.; Valenzano, R.; and Xie, F. 2011. Arvand: the art of random walks. *The 2011 International Planning Competition* 15.

Olsen, A., and Bryce, D. 2011. Randward and lamar: Randomizing the ff heuristic. *The 2011 International Planning Competition* 55.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *AAAI*, volume 8, 975–982.

Richter, S.; Westphal, M.; and Helmert, M. 2011. Lama 2008 and 2011. *The 2011 International Planning Competition* 50.

Rintanen, J. 2011. Madagascar: Efficient planning with sat. *The 2011 International Planning Competition* 61.

Roberts, M., and Howe, A. 2009. Learning from planner performance. *Artificial Intelligence* 173(5):536–561.

Vidal, V. 2011. Yahsp2: Keep it simple, stupid. *The 2011 International Planning Competition* 83–90.

Witten, I. H., and Frank, E. 2005. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Jasper: the Art of Exploration in Greedy Best First Search

Fan Xie and **Martin Müller** and **Robert Holte**
Computing Science, University of Alberta
Edmonton, Canada
{fxie2,mmueller,robert.holte}@ualberta.ca

Introduction

LAMA-2011 (Richter and Westphal 2010) is the clear winner of the sequential satisficing track in the latest International Planning Competition (IPC-2011). It finds a first solution by Greedy Best-First Search (GBFS), and then continues to improve solutions using restarting weighted A* (Richter, Thayer, and Ruml 2010). *Diverse Anytime Search (DAS)* (Xie, Valenzano, and Müller 2013) is a meta-algorithm designed for solution improvement. It takes an anytime planner and a post-processing system, and adds restarts and randomization for better quality search.

Jasper is a satisficing planner that builds on LAMA-2011. It adds two modifications. First, it replaces the GBFS algorithm in LAMA-2011 with an improved GBFS variant, called *Type Exploration based Greedy Best-First Search with Local Search (Type-GBFS-LS)*. GBFS always expands a node n that is closest to a goal state according to a heuristic h. GBFS' performance strongly depends on h. Uninformative or misleading heuristics can massively increase the time and memory complexity of such searches. Type-GBFS-LS is an improved version of GBFS that is less sensitive to such flaws in heuristic functions. Second, it implements the DAS system for solution improvement, which takes the modified LAMA-2011 as the anytime planner and Aras (Nakhost and Müller 2010) as the post-processing system.

A detailed description of the implementation of DAS can be found in the ICAPS paper by Xie, Valenzano and Müller (Xie, Valenzano, and Müller 2013). This paper focuses on describing the new search algorithm, Type-GBFS-LS.

The remainder of this paper is organized as follows. First, we motivate this work by discussing the two potential problems of GBFS: *uninformative heuristic region and misleading heuristics*, followed by describing two corresponding solutions as well as their combination, Type-GBFS-LS. Later, experimental results show that the proposed algorithms improve the state of the art planner LAMA-2011 significantly.

Uninformative Heuristic Regions (UHR) and GBFS with Local Search

The notion of an Uninformative Heuristic Region (UHR) includes both *local minima* and *plateaus*. A *local minimum* is a state with minimum h-value within a local region, which is not a global minimum. A *plateau* is an area of the state space where all states have the same heuristic value.

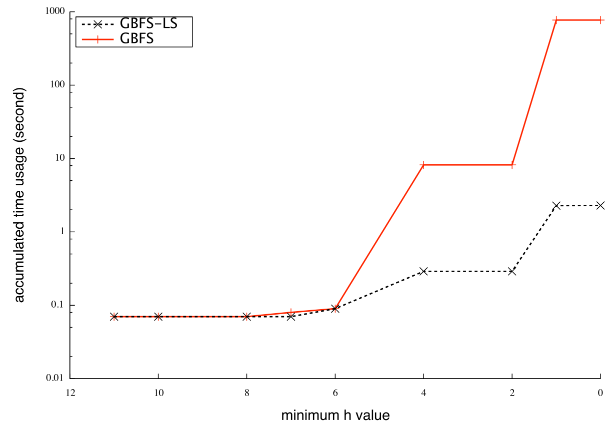


Figure 1: Cumulative search time (in seconds) of GBFS, and GBFS-LS with h^{FF} for first reaching a given h_{min} in 2004-notankage #21.

As an example, the IPC domain 2004-notankage has no dead ends, but contains large plateaus and local minima (Hoffmann 2011). Instance #21 shown in Figure 1 serves to illustrate a case of bad search behavior in GBFS due to UHRs. The figure plots the current minimum heuristic value h_{min} in the closed list on the x-axis against the log-scale cumulative search time needed to first reach h_{min}. The solid line is for GBFS with h^{FF}. The two huge increases in search time, with the largest (763 seconds) for the step from $h_{min} = 2$ to $h_{min} = 1$, correspond to times when the search is stalled in UHRs. Since the large majority of overall search time is used to inefficiently find an escape from UHRs, it seems natural to try switching to a secondary search strategy which is better at escaping.

GBFS with Local Search

The new algorithm of *Greedy Best-First Search with Local Search (GBFS-LS)* uses a local GBFS (*LS*) whenever a global GBFS (G-GBFS) seems stuck. If G-GBFS fails to improve its minimum heuristic value h_{min} for a fixed number

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

of node expansions, then GBFS-LS runs a small local GBFS for exploration from the best node n in a global-level open list.

LS shares the closed list of G-GBFS, but maintains its own separate open list *local_open* that is cleared before each local search. LS succeeds if it finds a new best node v with $h(v) < h_{min}$ before it exceeds a given limit on the number of nodes. In any case, the remaining nodes in *local_open* are merged into the global open list. A local search tree grown from a single node n is much more focused and grows deep much more quickly than the global open list in G-GBFS. It also restricts the search to a single plateau, while G-GBFS can get stuck when exploring many separate plateaus simultaneously. Both G-GBFS and *LS* use a first-in-first-out tie-breaking rule. A detailed description of GBFS-LS can be found in (Xie, Müller, and Holte 2014). In Figure 1, the same problem takes GBFS-LS only 1 second to solve, while it takes the basic GBFS around 1000 seconds.

Misleading Heuristics (ML) and Type Exploration in GBFS

Early mistakes are mistakes in search direction at shallow levels of the search tree, caused by sibling nodes being expanded in the wrong order due to a misleading heuristic. the root node of a *bad subtree*, which contains no solution or only hard-to-find solutions, has a lower heuristic value than a sibling which would lead to a quick solution.

The 2011-Nomystery domain from IPC-2011 is a typical example where delete-relaxation heuristics systematically make early mistakes (Nakhost, Hoffmann, and Müller 2012). In this transportation domain with limited non-replenishable fuel, delete-relaxation heuristics such as h^{FF} ignore the crucial aspect of fuel consumption, which makes the heuristic overoptimistic and misleading. Bad subtrees in the search tree, which over-consume fuel early on, are searched exhaustively, before any good subtrees which consume less fuel and can lead to a solution are explored. As a result, while the random walk-based planner Arvand with its focus on exploration solved 19 out of 20 nomystery instances in IPC-2011, LAMA-2011 solved only 10.

Previous exploration methods in GBFS suffer from biasing their exploration heavily towards the neighborhood of nodes in the open list. In the case of early mistakes, the large majority of these nodes is in useless regions of the search space. Consider the nodes in the regular h^{FF} open list of LAMA-2011 while solving the problem 2011-nomystery #12. Figure 2 shows snapshots of their h-value distribution after 2,000, 10,000 and 50,000 nodes expanded. In the figure, the x-axis represents different heuristic values and the y-axis represents the number of nodes with a specific h value in the open list. The solution eventually found by LAMA-2011 goes through a single node n in this 50,000 node list, with $h(n) = 18$. This node is marked with a star in the figure. Over 99% of the nodes in the open list have lower h-values, and will be expanded first, along with much of their subtrees. However, in this example, none of those nodes leads to a solution. The open list is flooded with a large number of very similar, useless nodes from undetected dead ends

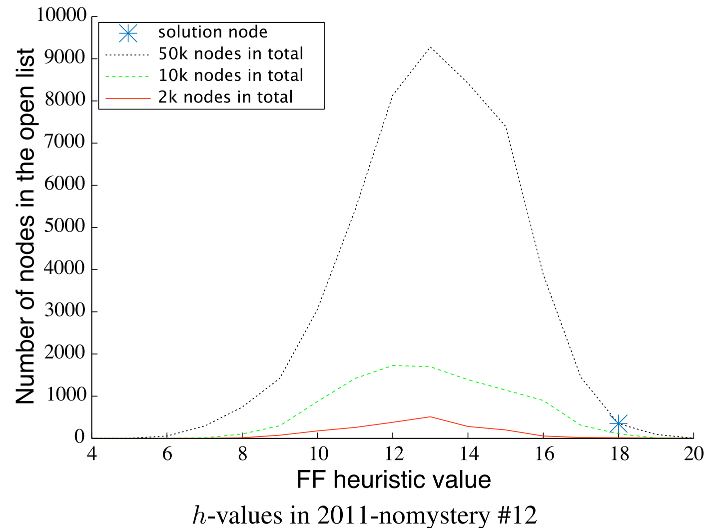


Figure 2: h-value distribution in the regular h^{FF} open list of LAMA-2011.

or local minima.

ϵ-GBFS (Valenzano et al. 2014) samples nodes uniformly over the whole open list. This is not too useful when entries are heavily clustered in bad subtrees. In the example above, ϵ-GBFS has a less than 1% probability to pick a node with h-value 18 or more in its exploration step, which itself is only executed with probability ϵ. Furthermore, the algorithm must potentially select several good successor nodes before making measurable progress towards a solution by finding an exit node with a lower h-value.

Type System

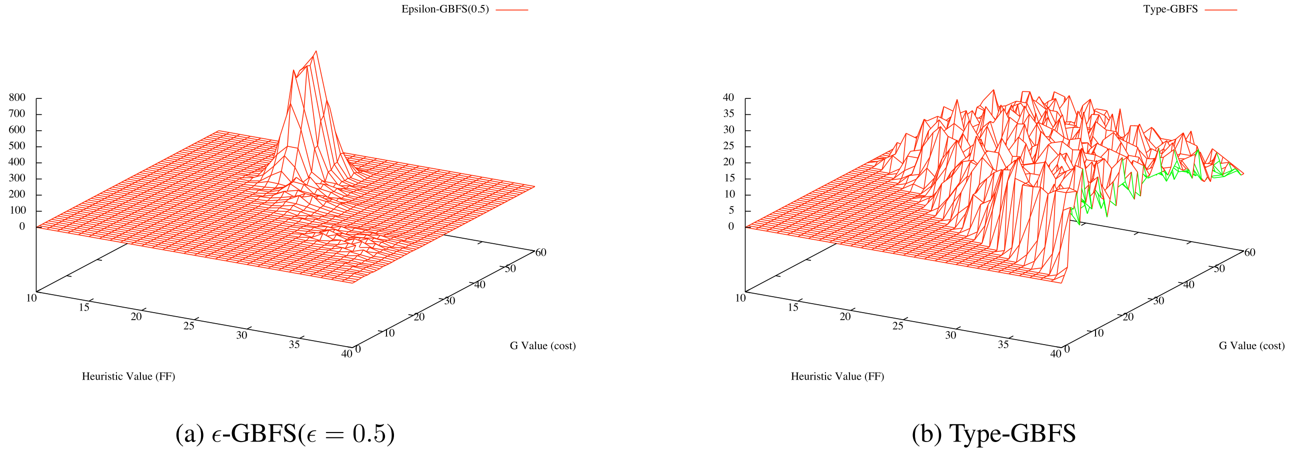
Can the open list be sampled in a way that avoids the over-concentration on a cluster of very similar nodes? A *type system* (Lelis, Zilles, and Holte 2013), which is based on earlier ideas of stratified sampling (Chen 1992), is one possible approach. It is defined as follows:

Definition 1 *(Lelis, Zilles, and Holte 2013) Let S be the set of nodes in search space. $T = \{t_1, \ldots, t_n\}$ is a type system for S if T is a disjoint partitioning of S. For every $s \in S$, $T(s)$ denotes the unique $t \in T$ with $s \in S$.*

Types can be defined using any property of nodes. The simple type system used here defines the type of a node s in terms of its h-value for different heuristics h, and its g-value. A simple and successful choice is the pair $T(s) = (h_{FF}(s), g(s))$. The intuition behind such type systems is that they can roughly differentiate between nodes in different search regions, and help explore regions differents from the nodes where GBFS gets stuck.

Type-GBFS: Adding a Type System to GBFS

Type-GBFS uses a simple two level *type bucket* data structure tb which organizes its nodes in buckets according to their type. Type bucket-based node selection works as follows: first, pick a bucket b uniformly at random from among



(a) ϵ-GBFS($\epsilon = 0.5$)

(b) Type-GBFS

Figure 3: Distribution of types over the first 20,000 nodes expanded by the exploring phase (ϵ-exploration or type buckets) of ϵ-GBFS($\epsilon = 0.5$) and Type-GBFS.

all non-empty buckets and then pick a node n uniformly at random from all nodes in b. Type-GBFS alternately expands a node from the regular open list O and from tb, and each new node is added to both O and tb. A detailed description of Type-GBFS can be found in (Xie et al. 2014).

Type-GBFS and ϵ-GBFS with $\epsilon = 0.5$ both spend half their search effort on exploration. However, the distribution of types of the explored nodes is very different. Figure 3 shows the frequency of explored node types for ϵ-GBFS with $\epsilon = 0.5$ and Type-GBFS[1] after 20,000 nodes in the same format. ϵ-GBFS mainly explores nodes close to the low heuristic value types, while Type-GBFS explores much more uniformly over the space of types.

Note that the z-axis scales are different for the two figures. The single most explored type contains around 800 nodes for ϵ-GBFS but only 40 for Type-GBFS. The presence or absence of exploration helps explain the relative performance in 2011-Nomystery. The coverage for the 20 instances of this domain for one typical run under IPC conditions is 9 for GBFS, 11 for ϵ-GBFS with $\epsilon = 0.5$, and 17 for Type-GBFS.

Combining GBFS-LS and Type-GBFS

GBFS-LS and Type-GBFS are designed for two different problems in GBFS. Jasper applies both enhancements to GBFS. The new algorithm is called *Type Exploration based Greedy Best-First Search with Local Search (Type-GBFS-LS)*. Like GBFS-LS, Type-GBFS-LS uses a local search when the global search gets stuck. However, it replaces GBFS with Type-GBFS in both the global level search and the local level search.

Experiments

Experiments were run on a set of 2112 problems in 54 domains from the seven International Planning Competitions, using one core of a 2.8 GHz machine with 4 GB memory and 30 minutes per instance. Results for planners which use randomization are averaged over five runs.

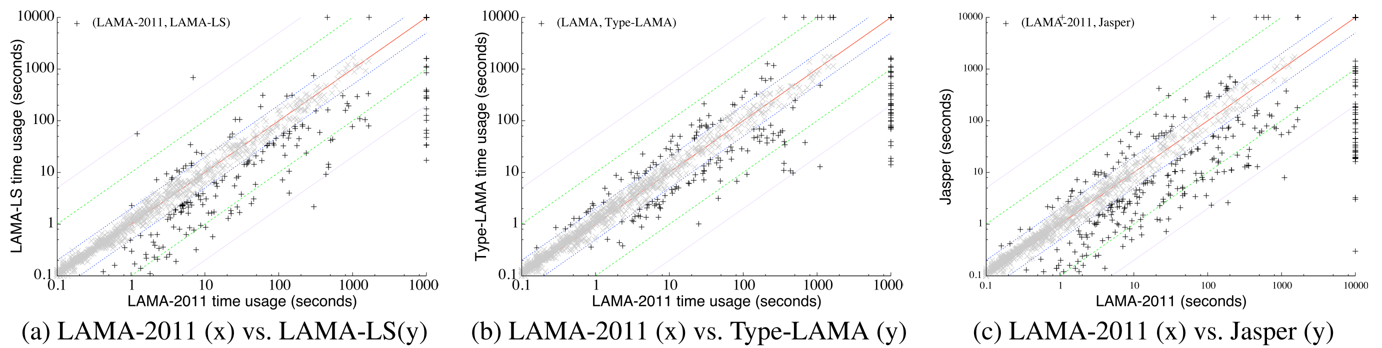
The performance comparison in this section includes the following planners:

- **LAMA-2011:** only the first iteration of LAMA using GBFS is run, with deferred evaluation, preferred operators and multi-heuristics (h^{FF}, h^{lm}) (Richter and Westphal 2010).
- **LAMA-LS:** Configured like LAMA-2011, but with GBFS replaced by GBFS-LS.
- **Type-LAMA:** With GBFS replaced by Type-GBFS, uses the same four queues as LAMA-2011, plus (h^{FF}, g) type buckets.
- **Jasper:** Configured like LAMA-2011, but with GBFS replaced by Type-GBFS-LS. It uses the same four queues as LAMA-2011 plus (h^{FF}, g) type buckets in both the global search and the local search.

Table 1 shows the coverage results for the four planners. All the three proposed planners get better results than LAMA-2011, with the best result of 1953.0 for Jasper.

Each diagram in Figure 4 compares one planner with LAMA-2011 on their time performance. Every data point represents one instance, with the search time for LAMA-2011 on the x-axis plotted against the corresponding planner on the y-axis. Only problems for which both algorithms need at least 0.1 seconds are shown. Points below the main diagonal represent instances that Type-GBFS solves faster than GBFS. For ease of comparison, additional reference lines indicate $2\times$, $10\times$ and $50\times$ relative speed. Data points within a factor of 2 are greyed out in order to highlight the instances with substantial differences. Problems that were only solved by one algorithm within the 1800 second time limit are included at $x = 10000$ and $y = 10000$.

[1]Some explored types are outside the (h, g) range shown in Figure 3 (b).



(a) LAMA-2011 (x) vs. LAMA-LS(y) (b) LAMA-2011 (x) vs. Type-LAMA (y) (c) LAMA-2011 (x) vs. Jasper (y)

Figure 4: Comparison of search time: LAMA-2011 vs. LAMA-LS (a), Type-LAMA(b) and Jasper.

All the three proposed planners show a clear overall improvement over LAMA-2011 in terms of speed. Jasper has the best overall performance. It solves more problems than LAMA-LS. Besides its advantage in coverage, it wins the time comparison with Type-LAMA for a larger number of instances by factors 2x and 10x.

Planner	LAMA-2011	LAMA-LS	Type-LAMA	Jasper
Coverage	2113	1931	1949.8	**1953.0**

Table 1: IPC coverage out of 2112.

References

Benton, J.; Haslum, P.; Helmert, M.; Katz, M.; and Thayer, J., eds. 2014. *Proceedings of the Sixth Workshop on Heuristic Search for Domain-Independent Planning, HSDIP 2014*.

Brafman, R. I.; Geffner, H.; Hoffmann, J.; and Kautz, H. A., eds. 2010. *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*.

Chen, P. C. 1992. Heuristic sampling: A method for predicting the performance of tree searching programs. *SIAM J. Comput.* 21(2):295–315.

Hoffmann, J. 2011. Where ignoring delete lists works, part II: Causal graphs. In Bacchus, F.; Domshlak, C.; Edelkamp, S.; and Helmert, M., eds., *ICAPS*, 98–105. AAAI.

Lelis, L. H. S.; Zilles, S.; and Holte, R. C. 2013. Stratified tree search: a novel suboptimal heuristic search algorithm. In Gini, M. L.; Shehory, O.; Ito, T.; and Jonker, C. M., eds., *AAMAS*, 555–562.

Nakhost, H., and Müller, M. 2010. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In Brafman et al. (2010), 121–128.

Nakhost, H.; Hoffmann, J.; and Müller, M. 2012. Resource-constrained planning: A Monte Carlo random walk approach. In McCluskey, L.; Williams, B.; Silva, J. R.; and Bonet, B., eds., *Proceeedings of the Twenty-Second International Conference on Automated Planning and Scheduling (ICAPS-2012)*, 181–189.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Richter, S.; Thayer, J.; and Ruml, W. 2010. The joy of forgetting: Faster anytime search via restarting. In Brafman et al. (2010), 137–144.

Valenzano, R.; Schaeffer, J.; Sturtevant, N.; and Xie, F. 2014. A comparison of knowledge-based GBFS enhancements and knowledge-free exploration. In *ICAPS*.

Xie, F.; Müller, M.; Holte, R.; and Imai, T. 2014. Type-based exploration with multiple search queues for satisficing planning. In Benton et al. (2014). 9 pages.

Xie, F.; Müller, M.; and Holte, R. 2014. Adding local exploration to greedy best-first search in satisficing planning. In Benton et al. (2014). 9 pages.

Xie, F.; Valenzano, R.; and Müller, M. 2013. Better time constrained search via randomization and postprocessing. In *ICAPS*, 269–277. AAAI.

Mercury Planner: Pushing the Limits of Partial Delete Relaxation

Michael Katz
IBM Haifa Research Labs
Haifa, Israel
katzm@il.ibm.com

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

Abstract

Mercury is a sequential satisficing planner that is based mainly on the red-black planning heuristic. Red-black planning is a systematic approach to partial delete relaxation, taking into account *some* of the delete effects: Red variables take the relaxed (value-accumulating) semantics, while black variables take the regular semantics. Prior work on red-black plan heuristics has identified a powerful tractable fragment requiring the *black causal graph* – the projection of the causal graph onto the black variables – to be a DAG; but all implementations so far use a much simpler fragment where the black causal graph is required to not contain any arcs at all. We close that gap here, and we design techniques aimed at making red-black plans executable, short-cutting the search. Mercury planner is entered into sequential satisficing and agile tracks of the competition.

Planner structure

Mercury planner is a sequential satisficing planner that is implemented in the Fast Downward planning system (Helmert 2006). It performs multiple iterations of heuristic search, starting with a fast and inaccurate greedy best-first search. Once a solution is found, next iterations run weighted A^*, gradually decreasing the weight parameter, similarly to the famous LAMA planning system (Richter and Westphal 2010). The cost of the best plan found so far is used in following iterations for search space pruning. Search algorithms are guided by the red-black heuristic (Katz, Hoffmann, and Domshlak 2013b; 2013a; Katz and Hoffmann 2013), breaking ties using the landmark count heuristic (Porteous, Sebastia, and Hoffmann 2001). In addition, preferred operators are obtained from each of those heuristics. For red-black heuristic, which is based on FF (Hoffmann and Nebel 2001), we decided to use here the preferred operators of FF heuristic. As the rest of the components are well known, in what follows, we describe in detail the main novelty of *Mercury*, red-black heuristic.

Introduction

The *delete relaxation*, where state variables accumulate their values rather than switching between them, has played a key role in the success of satisficing planning systems, e. g. (Bonet and Geffner 2001; Hoffmann and Nebel 2001; Richter and Westphal 2010). Still, the delete relaxation has well-known pitfalls, for example the fundamental inability to account for moves back and forth (as done, e. g., by vehicles in transportation). It has thus been an actively researched question from the outset how to take *some* deletes into account, e. g. (Fox and Long 2001; Gerevini, Saetti, and Serina 2003; Helmert 2004; Helmert and Geffner 2008; Baier and Botea 2009; Cai, Hoffmann, and Helmert 2009; Haslum 2012; Keyder, Hoffmann, and Haslum 2012). Herein, we continue the most recent attempt, *red-black planning* (Katz, Hoffmann, and Domshlak 2013b; 2013a; Katz and Hoffmann 2013) where a subset of *red* state variables takes on the relaxed value-accumulating semantics, while the other *black* variables retain the regular semantics.

Katz et al. (2013b) introduced the red-black framework and conducted a theoretical investigation of tractability. Following up on this (2013a), they devised practical *red-black plan heuristics*, non-admissible heuristics generated by repairing fully delete-relaxed plans into red-black plans. Observing that this technique often suffers from dramatic overestimation incurred by following arbitrary decisions taken in delete-relaxed plans, Katz and Hoffmann (2013) refined the approach to rely less on such decisions, yielding a more flexible algorithm delivering better search guidance.

The *black causal graph* is the projection of the causal graph onto the black variables only. Both Katz et al. (2013a) and Katz and Hoffmann (2013) exploit, in theory, a tractable fragment characterized by *DAG black causal graphs*, but confine themselves to *arc-empty black causal graphs* – no arcs at all – in practice. Thus current red-black plan heuristics are based on a simplistic, almost trivial, tractable fragment of red-black planning. We herein close that gap, designing *red-black DAG heuristics* exploiting the full tractable fragment previously identified. To that end, we augment Katz and Hoffmann's implementation with a DAG-planning algorithm (executed several times within every call to the heuristic function). We devise some enhancements targeted at making the resulting red-black plans executable in the real task, stopping the search if they succeed in reaching the goal.

Background

Our approach is placed in the *finite-domain representation (FDR)* framework. We introduce FDR and its delete-relaxation as special cases of red-black planning. A **red-**

black (RB) planning task is a tuple $\Pi = \langle V^{\mathsf{B}}, V^{\mathsf{R}}, A, I, G \rangle$. V^{B} is a set of *black state variables* and V^{R} is a set of *red state variables*, where $V^{\mathsf{B}} \cap V^{\mathsf{R}} = \emptyset$ and each $v \in V := V^{\mathsf{B}} \cup V^{\mathsf{R}}$ is associated with a finite domain $\mathcal{D}(v)$. The *initial state* I is a complete assignment to V, the *goal* G is a partial assignment to V. Each action a is a pair $\langle \mathsf{pre}(a), \mathsf{eff}(a) \rangle$ of partial assignments to V called *precondition* and *effect*. We often refer to (partial) assignments as sets of *facts*, i. e., variable-value pairs $v = d$. For a partial assignment p, $\mathcal{V}(p)$ denotes the subset of V instantiated by p. For $V' \subseteq \mathcal{V}(p)$, $p[V']$ denotes the value of V' in p.

A state s assigns each $v \in V$ a non-empty subset $s[v] \subseteq \mathcal{D}(v)$, where $|s[v]| = 1$ for all $v \in V^{\mathsf{B}}$. An action a is applicable in state s if $\mathsf{pre}(a)[v] \in s[v]$ for all $v \in \mathcal{V}(\mathsf{pre}(a))$. Applying a in s changes the value of $v \in \mathcal{V}(\mathsf{eff}(a)) \cap V^{\mathsf{B}}$ to $\{\mathsf{eff}(a)[v]\}$, and changes the value of $v \in \mathcal{V}(\mathsf{eff}(a)) \cap V^{\mathsf{R}}$ to $s[v] \cup \{\mathsf{eff}(a)[v]\}$. By $s[\![\langle a_1, \dots, a_k \rangle]\!]$ we denote the state obtained from sequential application of $a_1, \dots, a_k$. An action sequence $\langle a_1, \dots, a_k \rangle$ is a *plan* if $G[v] \in I[\![\langle a_1, \dots, a_k \rangle]\!][v]$ for all $v \in \mathcal{V}(G)$.

Π is a **finite-domain representation (FDR)** planning task if $V^{\mathsf{R}} = \emptyset$, and is a **monotonic finite-domain representation (MFDR)** planning task if $V^{\mathsf{B}} = \emptyset$. Plans for MFDR tasks (i. e., for delete-relaxed tasks) can be generated in polynomial time. A key part of many satisficing planning systems is based on exploiting this property for deriving heuristic estimates, via delete-relaxing the task at hand. Generalizing this to red-black planning, the **red-black relaxation** of an FDR task Π relative to V^{R} is the RB task $\Pi^{*+}_{V^{\mathsf{R}}} = \langle V \setminus V^{\mathsf{R}}, V^{\mathsf{R}}, A, I, G \rangle$. A plan for $\Pi^{*+}_{V^{\mathsf{R}}}$ is a **red-black plan** for Π, and the length of a shortest possible red-black plan is denoted $h^{*+}_{V^{\mathsf{R}}}(\Pi)$. For arbitrary states s, $h^{*+}_{V^{\mathsf{R}}}(s)$ is defined via the RB task $\langle V \setminus V^{\mathsf{R}}, V^{\mathsf{R}}, A, s, G \rangle$. If $V^{\mathsf{R}} = V$, then red-black plans are **relaxed plans**, and $h^{*+}_{V^{\mathsf{R}}}$ coincides with the optimal delete relaxation heuristic h^+.

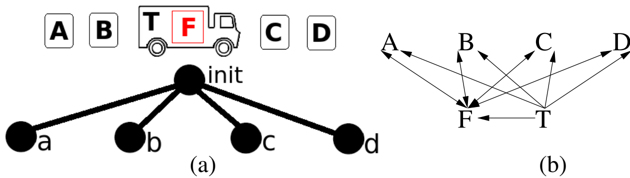


Figure 1: An example (a), and its causal graph (b).

In Figure 1, truck T needs to transport each package $X \in \{A, B, C, D\}$ to its respective goal location $x \in \{a, b, c, d\}$. The truck can only carry one package at a time, encoded by a Boolean variable F ("free"). A real plan has length 15 (8 loads/unloads, 7 drives), a relaxed plan has length 12 (4 drives suffice as there is no need to drive back). If we paint (only) T black, then $h^{*+}_{V^{\mathsf{R}}}(I) = 15$ as desired, but red-black plans may not be applicable in the real task, because F is still red so we can load several packages consecutively. Painting T *and* F black, that possibility disappears.[1]

Tractable fragments of red-black planning have been identified using standard structures. The **causal graph** CG_Π of Π is a digraph with vertices V. An arc (v, v') is in CG_Π if $v \neq v'$ and there exists an action $a \in A$ such that $(v, v') \in [\mathcal{V}(\mathsf{eff}(a)) \cup \mathcal{V}(\mathsf{pre}(a))] \times \mathcal{V}(\mathsf{eff}(a))$. The **domain transition graph** $\mathit{DTG}_\Pi(v)$ of a variable $v \in V$ is a labeled digraph with vertices $\mathcal{D}(v)$. The graph has an arc (d, d') induced by action a if $\mathsf{eff}(a)[v] = d'$, and either $\mathsf{pre}(a)[v] = d$ or $v \notin \mathcal{V}(\mathsf{pre}(a))$. The arc is labeled with its **outside condition** $\mathsf{pre}(a)[V \setminus \{v\}]$ and its **outside effect** $\mathsf{eff}(a)[V \setminus \{v\}]$.

The **black causal graph** $\mathit{CG}^{\mathsf{B}}_\Pi$ of Π is the sub-graph of CG_Π induced by V^{B}. An arc (d, d') is **relaxed side effects invertible**, **RSE-invertible** for short, if there exists an arc (d', d) with outside condition $\phi' \subseteq \phi \cup \psi$ where ϕ and ψ are the outside condition respectively outside effect of (d, d'). A variable v is RSE-invertible if all arcs in $\mathit{DTG}_\Pi(v)$ are RSE-invertible, and an RB task is RSE-invertible if all its black variables are. Prior work on red-black plan heuristics (Katz, Hoffmann, and Domshlak 2013a; Katz and Hoffmann 2013) proved that plan generation for RSE-invertible RB tasks with **DAG** (acyclic) black causal graphs is tractable, but used the much simpler fragment restricted to **arc-empty** black causal graphs in practice. In Figure 1, both T and F are RSE-invertible; if we paint only T black then the black causal graph is arc-empty, and if we paint both T and F black then the black causal graph is (not arc-empty but) a DAG.

Red-Black DAG Heuristics

Katz and Hoffmann (2013) provide an algorithm for RSE-invertible RB tasks with acyclic black causal graphs. To provide the context, Figure 2 shows Katz and Hoffmann's pseudo-code. The algorithm assumes as input the set R^+ of preconditions and goals on red variables in a fully delete-relaxed plan, i. e., $R^+ = G[V^{\mathsf{R}}] \cup \bigcup_{a \in \pi^+} \mathsf{pre}(a)[V^{\mathsf{R}}]$ where π^+ is a relaxed plan for Π. It then successively selects achieving actions for R^+, until all these red facts are true. Throughout the algorithm, *R denotes the set of red facts already achieved by the current red-black plan prefix π; B denotes the set of black variable values that can be achieved using only red outside conditions from R.*

For each action $a \in A'$ selected to achieve new facts from R^+, and for the global goal condition at the end, there may be black variables that do not have the required values. For example, say we paint T and F black in Figure 1. Then R^+ will have the form $\{A = T, A = a, B = T, B = b, C = T, C = c, D = T, D = d\}$. In the initial state, A' will contain only load actions. Say we execute $a = $load$(A, \mathit{init})$, entering $A = T$ into R and thus including unload(A, a) into A' in the next iteration. Trying to execute that action, we find that its black precondition $T = a$ is not satisfied. The call to ACHIEVE$(\{T = a\})$ is responsible for rectifying this.

ACHIEVE(g) creates a task Π^{B} over Π's black variables, asking to achieve g. As Katz and Hoffmann showed, Π^{B} is solvable, has a DAG causal graph, and has strongly connected DTGs (when restricting to actions a where $\mathsf{pre}(a) \subseteq I[\![\pi]\!]$). From this and Theorem 4.4 of Chen and Gimenez

[1] Indeed, all optimal red-black plans (but not some non-optimal ones) then are real plans. We will get back to this below: As we shall see, the ability to increase red-black plan applicability is a main advantage of our red-black DAG heuristics over the simpler red-black plan heuristics devised in earlier work.

Algorithm : REDBLACKPLANNING(Π, R^+)

```
main
  // Π = ⟨V^B, V^R, A, I, G⟩
  global R,  B ← ∅,  π ← ⟨⟩
  UPDATE()
  while R ⊉ R^+
    do { A' = {a ∈ A | pre(a) ⊆ B ∪ R, eff(a) ∩ (R^+ \ R) ≠ ∅}
         Select a ∈ A'
         if pre(a)[V^B] ⊈ I⟦π⟧
           then π ← π ∘ ACHIEVE(pre(a)[V^B])
         π ← π ∘ ⟨a⟩
         UPDATE() }
  if G[V^B] ⊈ I⟦π⟧
    then π ← π ∘ ACHIEVE(G[V^B])
  return π

procedure UPDATE()
  R ← I⟦π⟧[V^R]
  B ← B ∪ I⟦π⟧[V^B]
  for v ∈ V^B, ordered topologically by the black causal graph
    do B ← B ∪ DTG_Π(v)|_{R∪B}

procedure ACHIEVE(g)
  I^B ← I⟦π⟧[V^B]
  G^B ← g
  A^B ← {a^B | a ∈ A, a^B = ⟨pre(a)[V^B], eff(a)[V^B]⟩,
         pre(a) ⊆ R ∪ B, eff(a)[V^B] ⊆ B}
  ⟨a'_1^B, ..., a'_k^B⟩ ← an FDR plan for Π^B = ⟨V^B, A^B, I^B, G^B⟩
  return ⟨a'_1, ..., a'_k⟩
```

Figure 2: Red-black planning algorithm. $R^+ = G[V^{\mathsf{R}}] \cup \bigcup_{a \in \pi^+} \mathsf{pre}(a)[V^{\mathsf{R}}]$ where π^+ is a relaxed plan for Π.

(2010), it directly follows that a plan for Π^{B}, *in a succinct plan representation*, can be generated in polynomial time.

The "succinct plan representation" just mentioned consists of recursive macro actions for pairs of initial-value/other-value within each variable's DTG; it is required as plans for Π^{B} may be exponentially long. Chen and Gimenez' algorithm handling these macros involves the exhaustive enumeration of shortest paths for the mentioned value pairs in all DTGs, and it returns highly redundant plans moving precondition variables back to their initial value in between every two requests. For example, if a truck unloads two packages at the same location, then it is moved back to its start location in between the two unload actions.

Katz and Hoffmann (2013) shunned the complexity of DAG planning, and considered Π^{B} with arc-empty causal graphs, solving which is trivial. In our work, after exploring a few options, we decided to use the simple algorithm in Figure 3: Starting at the leaf variables and working up to the roots, the partial plan π^{B} is augmented with plan fragments bringing the supporting variables into place (a similar algorithm was mentioned, but not used, by Helmert (2006)).

Proposition 1 *The algorithm* DAGPLANNING(Π^{B}) *is sound and complete, and its runtime is polynomial in the size of* Π^{B} *and the length of the plan* π^{B} *returned.*

Note here that the length of π^{B} is worst-case exponential in the size of Π^{B}, and so is the runtime of

Algorithm : DAGPLANNING(Π^{B})

```
main
  π^B ← ⟨⟩
  for i = n downto 1
    do { // Denote π^B = ⟨a_1, ..., a_k⟩
         d ← I[v_i]
         for j = 1 to k
           do { π_j ← ⟨⟩
                if pre(a_j)[v_i] is defined
                  then { π_j ← π_{v_i}(d, pre(a_j)[v_i])
                         d ← pre(a_j)[v_i] } }
         π_{k+1} ← ⟨⟩
         if G[v_i] is defined
           then π_{k+1} ← π_{v_i}(d, G[v_i])
         π^B ← π_1 · ⟨a_1⟩ · ... · π_k · ⟨a_k⟩ · π_{k+1} }
  return π^B
```

Figure 3: Planning algorithm for FDR tasks Π^{B} with DAG causal graph $CG_{\Pi^{\mathsf{B}}}$ and strongly connected DTGs. $v_1, \dots, v_n$ is an ordering of variables V consistent with the topology of $CG_{\Pi^{\mathsf{B}}}$. $\pi_v(d, d')$ denotes an action sequence constituting a shortest path in $DTG_v(\Pi)$ from d to d'.

DAGPLANNING(Π^{B}). We trade the theoretical worst-case efficiency of Chen and Gimenez' algorithm against the practical advantage of not having to rely on exhaustive computation of shortest paths – anew for every call of DAGPLANNING, with "initial values" and DTGs from Π^{B} – for input tasks Π^{B} that typically have small plans (achieving the next action's black preconditions) anyhow.[2]

Unlike the macro-based algorithm of Chen and Gimenez, our DAGPLANNING algorithm does not superfluously keep switching supporting variables back to their initial values. But it is not especially clever, either: If variable v_0 supports two otherwise independent leaf variables v_1 and v_2, then the sub-plans for v_1 and v_2 will be inserted sequentially into π^{B}, losing any potential for synergies in the values of v_0 required.

Painting Strategy

Katz and Hoffmann explored a variety of *painting strategies*, i. e., strategies for selecting the black variables. We kept this simple here because, as we noticed, there actually is little choice, at least when accepting the rationale that we should paint black as many variables as possible: In most IPC domains, there are at most 2 possible paintings per task. To illustrate, consider Figure 1: We can paint T and F black, or paint T and the packages black. All other paintings either do not yield a DAG black causal graph, or are not set-inclusion maximal among such paintings. We thus adopted one of Katz and Hoffmann's basic strategies, namely ordering the variables by causal graph level, and iteratively painting variables red until the black causal graph is a DAG (Katz and

[2]One could estimate DAG plan length (e. g., using Helmert's (2006) causal graph heuristic), computing a red-black plan *length estimate* only. But that would forgo the possibility to actually execute DAG red-black plans, which is a key advantage in practice.

Hoffmann's original strategies continue until that graph is arc-empty).

Enhancing Red-Black Plan Applicability

One crucial advantage of red-black plans, over fully-delete relaxed plans, is that they have a much higher chance of actually working for the original planning task. This is especially so for the more powerful DAG red-black plans we generate here. In Figure 1, as already mentioned, if we paint just T black then the red-black plan *might* work; but if we paint both T and F black – moving to a non-trivial DAG black causal graph – then *every optimal red-black plan definitely works*. A simple possibility for exploiting this, already implemented in Katz and Hoffmann's (2013) earlier work, is to *stop search* if the red-black plan generated for a search state s is a plan for s in the original task.

There is a catch here, though – the red-black plans we generate are not optimal and thus are not guaranteed to execute in Figure 1. In our experiments, we observed that the red-black plans often were not executable due to simple reasons. We fixed this by augmenting the algorithms with the two following applicability enhancements.

(1) Say that, as above, $R^+ = \{A = T, A = a, B = T, B = b, C = T, C = c, D = T, D = d\}$ and REDBLACKPLANNING started by selecting load$(A, init)$. Unload(A, a) *might* be next, but the algorithm might just as well select load$(B, init)$. With T and F black, load$(B, init)$ has the black precondition $F = true$. Calling ACHIEVE$(\{F = true\})$ will obtain that precondition using unload$(A, init)$. Note here that variable A is red so the detrimental side effect is ignored. The same phenomenon may occur in any domain with renewable resources (like transportation capacity). We tackle it by giving a preference to actions $a \in A'$ getting whose black preconditions does not involve deleting R^+ facts already achieved beforehand. To avoid excessive overhead, we approximate this by recording, in a pre-process, which red facts may be deleted by moving each black variable, and prefer an action if none of its black preconditions may incur any such side effects.

(2) Our second enhancement pertains to the DTG paths chosen for the black precondition variables in DAGPLANNING (after REDBLACKPLANNING has already selected the next action). The red outside conditions are by design all reached (contained in R), but we can prefer paths whose red outside conditions are "active", i. e., true when executing the current red-black plan prefix in the real task. (E.g., if a capacity variable is red, then this will prefer loads/unloads that use the actual capacity instead of an arbitrary one.) In some special cases, non-active red outside conditions can be easily fixed by inserting additional supporting actions.

Supported Features

In contrast to previous years, a support for conditional effects is currently mandated. Since there is no straightforward adaptation of the red-black heuristics to the formalism that supports conditional effects, we have chosen here to compile them away. This was done by multiplying them out in the translation step. On one hand, this can lead to an exponential blow-up in the task representation size. On the other hand, it does not split up an operator application into a sequence of operator applications. Our decision was based on the speculation that the latter option could potentially decrease red-black plan applicability, one of the main advantages of the current red-black heuristics.

References

Baier, J. A., and Botea, A. 2009. Improving planning performance using low-conflict relaxed plans. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*. AAAI Press.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Cai, D.; Hoffmann, J.; and Helmert, M. 2009. Enhancing the context-enhanced additive heuristic with precedence constraints. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, 50–57. AAAI Press.

Chen, H., and Giménez, O. 2010. Causal graphs and structurally restricted planning. *Journal of Computer and System Sciences* 76(7):579–592.

Fox, M., and Long, D. 2001. Stan4: A hybrid planning strategy based on subproblem abstraction. *The AI Magazine* 22(3):81–84.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research* 20:239–290.

Haslum, P. 2012. Incremental lower bounds for additive cost planning problems. In Bonet, B.; McCluskey, L.; Silva, J. R.; and Williams, B., eds., *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, 74–82. AAAI Press.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS'08)*, 140–147. AAAI Press.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In Koenig, S.; Zilberstein, S.; and Koehler, J., eds., *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS'04)*, 161–170. Whistler, Canada: Morgan Kaufmann.

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

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

Katz, M., and Hoffmann, J. 2013. Red-black relaxed plan heuristics reloaded. In Helmert, M., and Röger, G., eds., *Proceedings of the 6th Annual Symposium on Combinatorial Search (SOCS'13)*, 105–113. AAAI Press.

Katz, M.; Hoffmann, J.; and Domshlak, C. 2013a. Red-black relaxed plan heuristics. In desJardins, M., and Littman, M., eds., *Proceedings of the 27th National Conference of the American Association for Artificial Intelligence (AAAI'13)*, 489–495. Bellevue, WA, USA: AAAI Press.

Katz, M.; Hoffmann, J.; and Domshlak, C. 2013b. Who said we need to relax *all* variables? In Borrajo, D.; Fratini, S.; Kambhampati, S.; and Oddi, A., eds., *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*, 126–134. Rome, Italy: AAAI Press.

Keyder, E.; Hoffmann, J.; and Haslum, P. 2012. Semi-relaxed plan heuristics. In Bonet, B.; McCluskey, L.; Silva, J. R.; and Williams, B., eds., *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*. AAAI Press.

Porteous, J.; Sebastia, L.; and Hoffmann, J. 2001. On the extraction, ordering, and usage of landmarks in planning. In Cesta, A., and Borrajo, D., eds., *Recent Advances in AI Planning. 6th European Conference on Planning (ECP-01)*, Lecture Notes in Artificial Intelligence, 37–48. Toledo, Spain: Springer-Verlag.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

NuCeLaR

Sergio Núñez and **Isabel Cenamor** and **Jesús Virseda**
Departamento de Informática, Universidad Carlos III de Madrid
Avda. de la Universidad, 30. 28911 Leganés (Madrid). Spain
sergio.nunez@uc3m.es, icenamor@inf.uc3m.es, jvirseda@inf.uc3m.es

Abstract

In this document we describe the techniques used to configure NuCeLaR, a sequential portfolio submitted and adapted to the three deterministic sequential tracks of the International Planning Competition 2014: sequential optimal, sequential satisficing and sequential multi-core. This portfolio has been configured using the combination of Machine Learning techniques and Mixed-Integer Programming.

Introduction

Since none of the existing planners dominates all others in every domains, the combination of some of them intuitively should improve their performance. Different approaches to combine existing planners have been proposed; i.e. using different components of different planners during the same search. Specifically, the combination of several planners independently executed in sequence with short timeouts are usually named portfolios. Works like (Helmert, Röger, and Karpas 2011; Gerevini, Saetti, and Vallati 2009) have shown that portfolios are a useful approach, since they achieved quite successful results in the previous International Planning Competitions (IPCs).

In this work, we apply a new strategy to combine existing planners using a portfolio approach: we configure a sequential portfolio for each kind of problem. To determine these kinds of problems, we apply machine learning to a set of planning problems from past IPCs. Particularly, we split these training problems in different groups using clustering techniques. These techniques are applied over a set of problem features extracted from the training problems. Once the set of training problems is split into different subsets (clusters), we compute a different portfolio configuration for each subset using a technique based on Mixed-Integer Programming (MIP) (Núñez, Borrajo, and Linares López 2012). Finally, we analyze the features of the problem to be solved and we run the corresponding portfolio configuration.

Figure 1 shows the two phases of the system: learning and deployment. The learning phase is also subdivided in two steps: clustering and portfolio generation.

The next sections describe in more detail both phases and provide specific information about the planners in the different tracks.

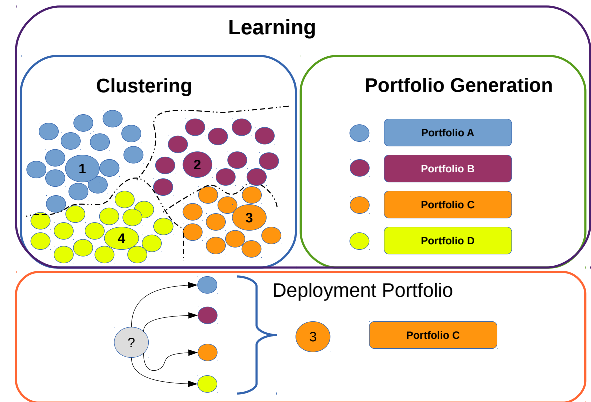


Figure 1: General System Diagram

Clustering Phase

The goal of data clustering (Jain 2010), also known as cluster analysis, is to discover the natural grouping of a set of patterns or points. Thus, this statistical classification technique determines whether the individuals of a population fall into a group or another by making quantitative comparisons of multiple characteristics. An operational definition of clustering can be defined as a representation of n objects, where the objective is to find k groups based on a measure of similarity. The similarities between instances that belong to the same group are high while the similarities between instances in different groups are low.

This methodology has fundamental challenges associated (Jain and Dubes 1988). The most relevant to achieve our objective are the feature selection, the data normalization, the number of clusters and whether the discovered clusters and partitions are valid for a portfolio configuration. We define an appropriate characterization of instances to find good solutions based in previous works (Roberts and Howe 2009; Cenamor, de la Rosa, and Fernández 2012; Virseda, Borrajo, and Alcázar 2013) and we evaluate them to obtain the best combination of features.

Therefore, we split the initial set of problems in several sub-groups using the selected features. However, the key is

to know how many groups is the best choice. To do that, we configure the corresponding portfolio for each possible number of clusters (within a range) and select the best one in order to find the best performance. Consequently, the selected k value is the one that solves more problems and achieves the best quality in the evaluation phase.

Portfolio Generator

The portfolio configuration for each cluster has been generated using MIP (Wolsey 2008), which computes the portfolio with the best achievable performance with respect to a selection of training planning tasks (Núñez, Borrajo, and Linares López 2012). The resulting portfolio configuration is a linear combination of candidate planners defined as a sorted set of pairs <planner, time>. The MIP model considers an *objective function* that maximizes a weighted sum of different parameters including: overall running time and quality.

Since the MIP model takes into account two different criteria (time and quality), it could be viewed and solved as a multi-objective maximization problem. Instead, we solve two MIP tasks in sequence while preserving the cost of the objective function from the solution of the first MIP. Specifically, we first run the MIP task to optimize only *quality* —i. e., sum of the plan quality of each solved problem for the satisficing track and the total number of solved planning tasks for the optimal track. If a solution exists, then a second execution of the MIP model is performed to find the combination of candidate planners that achieves the same quality (denoted as Q) while minimizing the overall running time. To enforce a solution with the same quality an additional constraint is added: $\sum_{i=0}^{n} quality_i \geq Q - \epsilon$, where ϵ is just any small real value used to avoid floating-point errors. Clearly, a solution is guaranteed to exist here, since a first solution was already found in the previous step. Algorithm 1 shows the steps followed to generate all the submitted portfolios where quality was maximized first, and then running time was minimized among the combinations that achieved the optimal quality. In our experiments, $\epsilon = 0.001$.

Algorithm 1 Build a portfolio optimizing quality and time

set weights to optimize only quality
portfolio$_1$:= solve the MIP task
Q := the resulting value of the objective function
if a solution exists **then**
  add constraint $\sum_{i=0}^{n} quality_i \geq Q - 0.001$
  set weights to optimize only overall running time
  portfolio$_2$:= solve the MIP task
  return *portfolio*$_2$
else
  exit with no solution
end if

Implementation of the Portfolio

Every submitted portfolio runs a fixed portfolio configuration for each problem to be solved. However, the runtime assigned to each component planner can change in unexpected ways during its execution when the component planner finishes prematurely: planner bugs, terminating cleanly without solving the instance, running out of memory, etc. Therefore, the total runtime of the executed portfolio can be lower than the available time. In this case, the submitted portfolio will run a default planner using the remaining time (RT). This default planner is picked up among the set of candidate planners which had a remarkable performance in the IPC 2011.

Sequential Optimization Track

In the design of NUCELAR, we have used all the problems defined in the optimal track of the IPC 2011. Also, we considered all the participant planners in that competition but FORKINIT, IFORKINIT and LMFORK because the organizers of the IPC 2014 had problems with the license of the Mosek LP solver [1]. Since the set of candidate planners was too small, we discarded the participant portfolios and added their component planners instead. Moreover, we included all the planners considered in the design of FDSS (Helmert, Röger, and Karpas 2011).

Table 1 shows the configuration of the NUCELAR portfolio for the sequential optimization track. This configuration is composed of six portfolio configurations in turn, one for each cluster, since we defined six clusters in the clustering phase. The execution sequence of the component planners has been sorted by increasing order of the allotted time.

Planner	Allotted time (s)
GAMER	1800
CPT4	550
RHW LANDMARKS	598
M&S-BISIM 1	652
FD AUTOTUNE	191
M&S-BISIM 2	326
GAMER	1282
LM-CUT	105
M&S-BISIM 1	188
M&S-BISIM 2	220
GAMER	1237
SELMAX	1800
CPT4	131
M&S-BISIM 2	331
h^{max} LANDMARKS	1356
SELMAX	RT

Table 1: Configuration of NUCELAR for the sequential optimization track.

Sequential Satisficing Track

NUCELAR for the satisficing track has been configured applying our technique over all the satisficing planning tasks

[1] MOSEK is a tool for solving mathematical optimization problems. http://mosek.com/

defined for the IPC 2011. Also, we have used all the participant planners in that competition and the component solvers of the participant portfolios. Moreover, we included all the planners considered in the design of FDSS. The FDSS planners are defined by a search algorithm, an evaluation method and a set of heuristics. Specifically, FDSS only considered weighted-A* w=3 (WA*) and greedy best-first search (GBFS), with EAGER (standard) and LAZY (deferred evaluation) variants of both search algorithms. Also, only four heuristics were considered: Additive heuristic ADD (Bonet and Geffner 2001), FF/additive heuristic FF (Hoffmann and Nebel 2001; Keyder and Geffner 2008), Causal Graph heuristic CG (Helmert 2004), and Context-Enhanced Additive heuristic CEA (Helmert and Geffner 2008).

The resulting portfolio is shown in Table 2, which contains one portfolio configuration for each one of the six clusters defined in the cluster analysis.

Planner	Allotted time (s)
YAHSP2 MT	2
LAMA 2011	3
FD AUTOTUNE 2	4
MADAGASCAR P	5
FD AUTOTUNE 1	5
YAHSP2	6
DAE YAHSP	27
ROAMER	30
GBFS - EAGER - FF, CG	78
GBFS - EAGER - CG	109
GBFS - LAZY - CG	116
WA* - LAZY - CG	227
PROBE	339
ARVAND	849
LAMA 2011	218
GBFS - LAZY - FF, CG	295
GBFS - LAZY - ADD, FF	1287
WA* - LAZY - FF	1800
FD AUTOTUNE 2	762
RANDWARD	1037
YAHSP2	5
YAHSP2 MT	5
FDSS 2	48
LAMA 2008	73
GBFS - EAGER - CG	142
FD AUTOTUNE 2	280
LAMA 2011	432
FORKUNIFORM	813
LAMAR	47
WA* - LAZY - FF	91
FDSS 1	1660
ROAMER	RT

Table 2: Configuration of NUCELAR for the sequential satisficing track.

Sequential Multi-Core Track

The NUCELAR portfolio for the multi-core track has been configured using the same training data (candidate planners and training planning tasks) and the same technique (adding the concept of core processor to the MIP model) used to configure the sequential satisficing portfolio.

The resulting portfolio is shown in Table 3, which is composed of six portfolio configurations since we defined six clusters in the clustering phase. Each portfolio configuration uses the four cores available and respects the wall-clock time limit defined in the competition.

Planner	Allotted Time (s)
PROBE	1356
ARVAND	1800
YAHSP2 MT	8
LAMA 2011	12
FD AUTOTUNE 2	16
MADAGASCAR P	20
FD AUTOTUNE 1	20
YAHSP2	24
DAE YAHSP	108
ROAMER	120
GBFS - EAGER - FF, CG	312
GBFS - EAGER - CG	436
GBFS - LAZY - CG	464
WA* - LAZY - CG	908
LAMA 2011	1800
GBFS - LAZY - FF, CG	1800
GBFS - LAZY - ADD, FF	1800
PROBE	1800
WA* - LAZY - FF	1800
PROBE	1800
LAMA 2011	1800
ARVAND	1800
FD AUTOTUNE 2	1800
RANDWARD	1800
PROBE	1800
LAMA 2011	1800
FORKUNIFORM	1800
LAMA 2011	1728
YAHSP2	20
YAHSP2 MT	20
FDSS 2	192
LAMA 2008	292
GBFS - EAGER - CG	142
FD AUTOTUNE 2	1120
LAMAR	1800
WA* - LAZY - FF	1800
FDSS 1	1800
PROBE	1800
LAMAR	RT
LAMA 2011	RT

Table 3: Configuration of the NUCELAR portfolio for the multi-core track.

Acknowledgments

We automatically generated sequential portfolios of existing planners in other competitions. Thus, we would like to acknowledge and thank the authors of the individual planners for their contribution and hard work.

This work has been partially supported by the Spanish project TSI-090302-2011-6 and the Planinteraction project TIN2011-27652-C03-02.

References

Bonet, B., and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence* 129(1-2):5–33.

Cenamor, I.; de la Rosa, T.; and Fernández, F. 2012. Mining ipc-2011 results. In *Proceedings of the Third Workshop on the International Planning Competition*.

Gerevini, A.; Saetti, A.; and Vallati, M. 2009. An Automatically Configurable Portfolio-based Planner with Macroactions: PbP. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, (ICAPS 2009)*. AAAI.

Helmert, M., and Geffner, H. 2008. Unifying the Causal Graph and Additive Heuristics. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 140–147.

Helmert, M.; Röger, G.; and Karpas, E. 2011. Fast Downward Stone Soup: A Baseline for Building Planner Portfolios. *In ICAPS 2011 Workshop on Planning and Learning* 28–35.

Helmert, M. 2004. A Planning Heuristic Based on Causal Graph Analysis. In *Proceedings of the Fourteenth International Conference 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 (JAIR)* 14:253–302.

Jain, A. K., and Dubes, R. C. 1988. *Algorithms for clustering data*. Prentice-Hall, Inc.

Jain, A. K. 2010. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters* 31(8):651–666.

Keyder, E., and Geffner, H. 2008. Heuristics for Planning with Action Costs Revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, 588–592.

Núñez, S.; Borrajo, D.; and Linares López, C. 2012. Performance Analysis of Planning Portfolios. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS, Niagara Falls, Ontario, Canada, July 19-21, 2012*. AAAI Press.

Roberts, M., and Howe, A. 2009. Learning from planner performance. *Artificial Intelligence* 173(5):536–561.

Virseda, J.; Borrajo, D.; and Alcázar, V. 2013. Learning heuristic functions for cost-based planning. *Planning and Learning* 6.

Wolsey, L. A. 2008. Mixed integer programming. *Wiley Encyclopedia of Computer Science and Engineering*.

Planets: A Planner for Net Benefit

Jonathan Teutenberg
Independent Researcher
jono.teutenberg@gmail.com

Abstract

Planets is a forward searching STRIPS state-based branch-and-bound best-first heuristic planner for satisficing planning problems with numeric values, soft goals and metric objective functions. Designed for narrative generation, Planets trades time for plan quality by generating multiple delete-relaxed solutions to closely approximate the optimal delete-relaxed heuristic h^+
The search uses an unweighted heuristic, however Planets strongly prefers actions that are helpful to at least one of its relaxed plans. Other notable features are a simple symmetry breaking for ordering actions of independent sub-plans; the use of occasional greedy probes in search of a backup plan; and a top-down approach to the selection of a subset of soft goals to achieve.

Background

Planets is used as a base-planner for narrative generation using an IMPRACTical (Teutenberg and Porteous 2013) approach. This presents planning problems for which

- Adding actions to a partial plan requires a complete state.
- Determining applicable actions in a state is very time-consuming, requiring the solving of a large number of relaxed plans.
- High quality – though not necessarily optimal – plans are required.
- Plan quality is a mixture of hard goals and a numeric objective function.

Originally Planets was entered in the satisficing preferences track of the IPC which was since cancelled. In this paper we still describe and briefly evaluate Planets' approach to the selection of soft goals to achieve from each state.

Due to the requirement for high quality plans, a best-first branch and bound with an unweighted heuristic is used, as opposed to a greedy search strategy or weighted a-star. The substantial cost required to determine applicable actions for narrative generation dominates the cost of heuristic evaluation. For this reason Planets attempts to approximate the costly h^+ heuristic by using the lowest total cost of many diverse relaxed plans. The full procedure is presented in the following section. While the approximation means this may not be an admissible heuristic, it is less likely to overestimate than the similar h_{FF} that extracts a single relaxed plan, or h_{add}. In addition, multiple explicit relaxed plans provide information on possible alternative solutions which Planets makes use of by treating the union of several of the best relaxed plans' helpful actions as being helpful in any given state.

Though there is as yet no standard planning domain for narratives, we are able to identify some forms of symmetry we expect to be present in high quality narratives. In particular Planets addresses the symmetry present in totally ordered plans that contain parallel independent sub-plans, as described in (Long and Fox 2003). Such sub-plans occur when two or more characters are active in their own sub-plots that only occasionally cross paths. In later sections this paper discusses the symmetry breaking procedure that enforces a single linearisation of interleaved sub-plans when their actions are identified as being helpful.

Finally, this paper presents a search for a low quality backup plan using a greedy best-first search through a sequence of landmarks, similar to Probe (Lipovetzky and Geffner 2011), using 10% of total planning time.

Planets is still a work in progress The latest version can be retrieved from *http://github.com/jteutenberg/Planets*.

Heuristic: Multiple Relaxed Plans

The heuristic for Planets' search uses solutions to the delete-relaxed version of the domain in the same vein as FF (Hoffmann and Nebel 2001). These relaxed plans are produced not through a relaxed planning graph, but by an expansion using h_{max} as described in (Keyder and Geffner 2008) as FF(h_a).

The notable difference in Planets is that it extracts multiple plans in parallel. When determining the action with best support for an open precondition, instead of selecting a single action a new partial relaxed plan is created for each possible supporting action. Once a limit on the number of partial relaxed plans is reached (set to 50 for the IPC) Planets completes each plan using greedy selection h_{max} criteria as in FF(h_a). This ensures that a solution found by FF(h_a) is also guaranteed to be considered by Planets.

Some care must be taken when extracting multiple solutions. When selecting supporting actions for an open fact

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

only the lowest h_{max} action is guaranteed to be reachable prior to this fact being achieved. To avoid cycles in the plan such as when two actions support one another, we restrict the extraction of supporting actions to those that were reached prior the best supporting action's expansion.

Another property that distinguishes Planets' relaxed plans is that they are an ordered sequence of actions (rather than a set), sorted in the order in which they were reached during the expansion using h_{max}. This sequence becomes important when selecting soft goals as described in later sections.

When no soft goals or metric objective functions are provided, the heuristic value for a state is the lowest total cost amongst all plans extracted.

Helpful Actions

A helpful action is an action that is applicable in a state and is a member of the set of actions that comprise a relaxed solution. Where planners using h_{FF} have one such solution, Planets maintains a set of solutions – in the IPC this is set to the best 5 plans from the 50 being extracted.

Helpful actions are therefore any applicable action that is a member of at least one of the relaxed plans. The idea here is that such actions represent progress along *some* path through the space of partial plans that leads towards a solution, even if it is not necessarily progress toward the most promising solution.

Planets maintains an open list partial plans that have been produced by helpful actions, and a set of plans produced by non-helpful actions. Plans from the open list are always used until it empties, at which point the set of non-helpful plans is emptied into the open list and search resumes.

Handling Preferences

The preferences track of the IPC was to include metric objective functions such that each goal $g \in G$ added a fixed, positive value and removed the total cost of actions. Planets solves is set up to solve a slightly more general class of metrics, of the form

$$m(S, \pi) = benefit(S \cap G) - totalCost(\pi)$$

where $benefit$ is an arbitrary function over the set of goal facts present in a state.

The selection of which goals to achieve is made during heuristic evaluation, where π is the current partial plan followed by the relaxed plan extending this to the goals. This follows other approaches that select subsets of goals to achieve based on relaxed plans such as (Smith 2004; Nigenda and Kambhampati 2005; García-Olaya, De La Rosa, and Borrajo 2011) Most of these use a bottom-up approach – incrementally adding most beneficial goals. Planets instead selects goals top-down – by incrementally selecting a set of goals to remove.

When selecting soft goals, Planets first creates relaxed plans that achieve all reachable goals using the approach described earlier. It then attempts to find the subset of actions, and the goals that result from them, for each relaxed plan such that the objective metric function is maximised using only these actions.

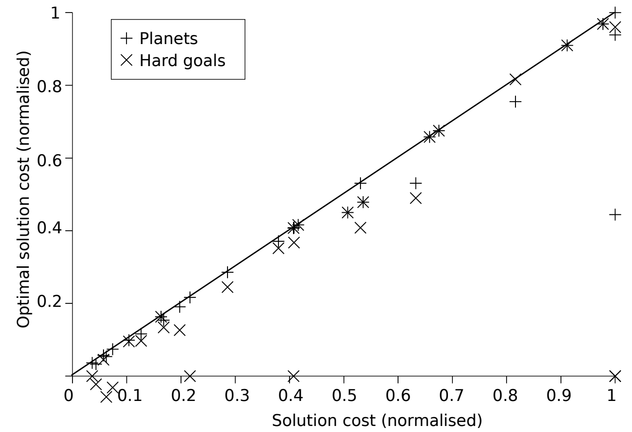


Figure 1: Ratio of optimal plan net benefit to Planets' solutions' net benefit, and to treating all goals as hard goals, over three planning domains.

Given a relaxed plan $\pi = a_1, a_2 \ldots a_n$, the cause of a fact p is defined as

$$cause(p) = a_i \text{ s.t. } p \in add(a_i) \wedge p \notin \bigcup_{j=1}^{i-1} add(a_j)$$

and the relevant actions to a fact p as

$$rel(p) = \{cause(p)\} \cup \{rel(q) \mid q \in pre(cause(p))\}$$

In particular, Planets uses this definition to generate the sets of relevant actions to each goal. Planets also makes use of the inverse of this relationship, the set of all goals that each action a_i is relevant to:

$$goals(a_i) = \{g \in G \mid a_i \in rel(g)\}$$

There are $2^{|G|}$ subsets of goals to consider for removal. However we can use the relevant actions defined above to reduce the partitions that need to be considered. If a subset of goals $G_{out} \subseteq G$ are to not be achieved then an action a_i will only be removed if it no longer has any goals that it is relevant to, or $goals(a_i) \subseteq G_{out}$. Planets therefore constructs the set of unique subsets of goals that can have some effect on the relaxed plan $\bigcup_{i=1}^{n} \{goal(a_i)\}$ and only considers these n candidates for removal in an iterative process.

Figure 1 shows results of goal selection on three domains from the preferences track of IPC 2008: crew-planning, transport-numeric and elevators-numeric, with values normalised by the highest benefit plan in their domain. The comparison against treating all goals as hard goals falls down on other domains in which these problems become unsolvable. In 28% of problems the hard goal version failed to produce a higher quality than the empty plan; in a further 34% the preference selection improved plan quality; and in 1 case it performed worse.

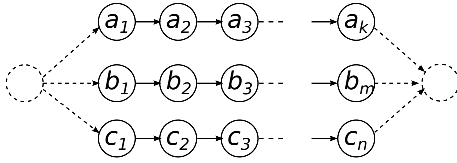


Figure 2: An idealised example of a portion of a solution plan with independent sub-plans. Circles represent actions and arrows a causal link from effect to precondition.

Simple Symmetry Elimination

When searching the space of totally ordered partial plans many redundant interleavings, or linearisations, of independent sub-plans will be considered. Two sub-plans A and B are independent when $\forall a \in A, \forall b \in B$ a and b are non-interfering, using the definition of interference from (Long and Fox 2003). An idealised example is shown in Figure 2 where three independent sub-plans that are each simple chains of causality make up a section of a solution.

The number of partial plans considered by a brute-force search through such a region is in the order of the number of unique states produced by all possible linearisations of the sub-plans (as the search is automatically terminated when a duplicate state is encountered). This is in the order of the product of the lengths of the sub-plans – kmn in the example shown. However, when all of these actions form part of the final solution, all linearisations result in the same state and so only one linearisation of the actions needs to be considered. A search through the single linearisation produces only up to $k + n + m$ unique states.

Planets applies symmetry breaking when a relaxed plan has two or more independent helpful actions H, up to a maximum of four actions. It forces an early commitment to the inclusion (or exclusion) of each of these actions. This begins by creating new partial plans for every selection $T \subset H$ of actions from this set (up to 2^4 such plans).

For a subset T all $a \in T$ are added to the partial plan in arbitrary order. In addition to the plan, each search node also contains an action blacklist to which all $b \in H \setminus T$ are added. Actions in the blacklist are not applicable in any state, and are not used during relaxed plan construction. An action is removed from the blacklist when an action is added to the plan that meets the requirements to interfere with it.

On the transport-numeric problems from IPC 2008, when there are at least two trucks and two packages (so the possibility of narrative-like independent sub-plots exist) this symmetry breaking reduces the number of evaluated nodes by an average of 14%. On those problems with a single truck there is no difference in the search space explored.

Probing for Backup Plans

While the planning proper continues, it may be necessary to buy some time by presenting actions from the beginning of a valid, if low-quality, backup plan. For this reason we have included a greedy best-first search inspired by PROBE (Lipovetzky and Geffner 2011) that takes over search for 10% of wall-clock time.

The rough procedure used for the backup planning is:

- Select as a start state the next best state in the branch-and-bound open list
- Generate a sequence of landmarks from this state to the goals
- Search to each landmark in turn based on heuristic values only with a beam search (width set to 50 for the IPC)

In comparison to PROBE, this implementation is missing the notion of commitments when selecting landmarks, but most significantly does not yet enforce all goal facts to be true simultaneously. This reduces the likelihood of successfully finding a backup plan in many domains.

Heuristic estimates and their associated relaxed plans that are generated during probes are stored. When the primary branch-and-bound search reaches such a state visited by a probe this is then re-used. The probes themselves will never expand a state previously visited by an earlier probe or by the primary search.

References

García-Olaya, A.; De La Rosa, T.; and Borrajo, D. 2011. Using the relaxed plan heuristic to select goals in oversubscription planning problems. In *Proceedings of the 14th International Conference on Advances in Artificial Intelligence: Spanish Association for Artificial Intelligence*.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation throughHeuristic Search. *Journal of AI Research* 14:253–302.

Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *In Proc. ECAI 2008*.

Lipovetzky, N., and Geffner, H. 2011. Searching for plans with carefully designed probes. In *Proc. of the 21st International Conference on Automated Planning and Scheduling (ICAPS)*, 154–161.

Long, D., and Fox, M. 2003. Plan permutation symmetries as a source of inefficiency in planning. In *22nd UK Planning and Scheduling Special Interest Group*.

Nigenda, R. S., and Kambhampati, S. 2005. Planning graph heuristics for selecting objectives in over-subscription planning problems. In *Proc. of the 15th Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Smith, D. 2004. Choosing objectives in over-subscription planning. In *Proc. of the 14th Int. Conf. on Automated Planning and Scheduling (ICAPS 2004)*.

Teutenberg, J., and Porteous, J. 2013. Efficient Intent-Based Narrative Generation Using Multiple Planning Agents. In *Proc. of 13th Int. Conf. on Autonomous Agents and MultiAgent Systems (AAMAS-13)*.

RPT: Random Planning Tree

Vidal Alcázar, Susana Fernández, Daniel Borrajo
Universidad Carlos III de Madrid
Av. Universidad, 30
28911 Leganés, Spain
valcazar@inf.uc3m.es;sfarregu@inf.uc3m.es;dborrajo@ia.uc3m.es

Manuela Veloso
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh PA 15213-3890, USA
mmv@cs.cmu.edu

Abstract

Rapidly-exploring random trees (RRTs) are data structures and search algorithms designed to be used in continuous path planning problems. They are one of the most successful state-of-the-art techniques in motion planning, as they offer a great degree of flexibility and reliability. Random Planning Tree (RPT) is a planner that implements RRTs for their use in automated planning.

Introduction

Single-query motion planning and satisficing planning have many points in common. However, bringing techniques from one area to the other is not straightforward. The main difference between the two areas is the defining characteristics of the search space. In motion planning, the original search space of these problems is an euclidean explicit continuous space, whereas in automated planning the search space is a multi-dimensional implicit discrete space. This has lead to both areas being developed without much interaction despite the potential benefits of an exchange of knowledge between the two communities.

The presented planner, Random Planning Tree (RPT), tries to bridge the gap between the two areas by proposing the use of an RRT in automated planning. The motivation is that RRTs may be able to overcome some of the shortcomings that forward search planners have while keeping most of their good properties. This planner builds on previous work by the same authors (Alcázar, Veloso, and Borrajo 2011); the main difference is the use of a broader range of state invariants in the sampling process.

Background

In this section we will present RRTs and the first planner inspired by RRTs, RRT-Plan (Burfoot, Pineau, and Dudek 2006). Regarding RRTs, this includes both the original definition as a data structure and its subsequent evolution as a single-query search algorithm in motion planning.

Rapidly-exploring Random Trees

RRTs (LaValle and Kuffner 1999) were proposed as both a sampling algorithm and a data structure designed to allow fast searches in high-dimensional spaces in motion planning. RRTs are progressively built towards unexplored regions of the space from an initial configuration. Configurations describe the position, orientation and velocity of the movable objects in motion planning and are equivalent to states in other search applications.

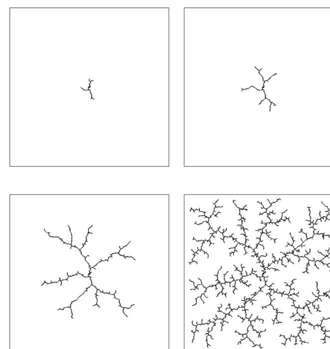


Figure 1: Progressive construction of an RRT.

At start, the algorithm creates a tree containing the initial configuration. At every step, a random q_{rand} configuration is chosen from all the configuration space and for that configuration the nearest configuration already in the tree q_{near} is computed. For this a definition of distance is required (in motion planning the euclidean distance is usually chosen as the distance measure). When the nearest configuration is found, a local planner tries to join q_{near} with q_{rand} with a limit distance ε. If q_{rand} was reached, it is added to the tree and connected with an edge to q_{near}. If q_{rand} was not reached, then the configuration q_{new} obtained at the end of the local search is added to the tree in the same way as long as there was no collision with an obstacle during search. In the search literature, the term *local search* refers to search algorithms that do not keep track of all the states that they have visited. The most representative algorithm of this kind is Hill Climbing, although many others exist. Here, though, whenever we use the term *local search* we mean the process of solving the subproblem needed to create a new branch of the tree. This operation is called the *Extend* step, illustrated in Figure 2. This process is repeated until some criteria is met, like a limit on the size of the tree. Algorithm 1 gives an outline of the process.

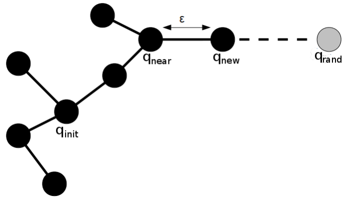


Figure 2: Extend phase of an RRT.

Algorithm 1: Description of the building process of an RRT.

Data: Search space S, initial configuration q_{init}, limit ε, ending criteria *end*
Result: RRT *tree*
begin
  $tree \longleftarrow q_{init}$
  while $\neg$ *end* **do**
    $q_{rand} \longleftarrow sampleSpace(S)$
    $q_{near} \longleftarrow nearest(tree, q_{rand}, S)$
    $q_{new} \longleftarrow join(q_{near}, q_{rand}, \varepsilon, S)$
    if reachable(q_{new}) **then**
      $addConfiguration(tree, q_{near}, q_{new})$
  return *tree*
end

Once the RRT has been built, multiples queries can be issued. For each query, the nearest configurations (node) of the tree to both the initial and the goal configurations of the query are found. Then, the initial and final configurations are joined to the tree to those nearest configurations using the local planner and a path is retrieved by tracing back edges through the tree structure.

The key advantage of RRTs is that they are intrinsically biased towards regions with a low density of configurations in their building process. This can be explained by looking at the Voronoi diagram at every step of the building process. The Voronoi diagram is conformed by Voronoi regions; Voronoi regions associated to a given node q of the tree are areas such that every point in the area is closer to q than to any other node q' of the RRT. The Voronoi region of a given node is larger when the area around that node has not been explored. This way, the probability of a configuration being sampled in an unexplored region is higher as larger Voronoi regions will be more likely to contain the sampled configuration (Aurenhammer 1991). This has the advantage of naturally guiding the tree by extending nodes at the edge of unexplored regions with a higher probability while just performing uniform sampling. Besides, the characteristics of the Voronoi diagram are an indicative of the adequateness of the tree. For example, a tree whose Voronoi diagram is formed by regions of similar size covers uniformly the search space, whereas large disparities in the size of the regions mean that the tree may have left big areas of the search space unexplored. Apart from this, another notable characteristic is that RRTs are probabilistically complete, as they will cover the whole search space if the number of sampled configurations tends to infinity. Figure 3 shows the Voronoi diagrams of the RRTs previously shown in Figure 1.

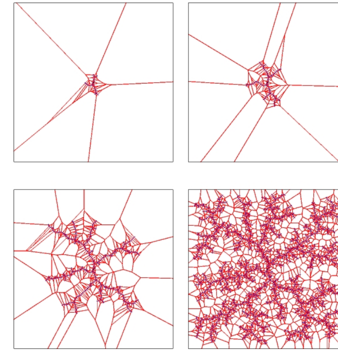


Figure 3: Voronoi Diagram of an RRT.

RRT-Connect

After corroborating how successful RRTs were for multi-query motion planning problems, researchers in motion planning realized that using multi-query RRTs was often more efficient and robust than using specific single-query motion planning algorithms even for a single query. Motivated by this fact and aiming to develop a more suitable RRT-like algorithm for the single-query case, a variation for single-query problems called RRT-Connect was proposed (Kuffner and LaValle 2000). The modifications introduced were the following:

- Two trees are grown at the same time by alternatively expanding them. The initial configuration of the trees are the initial and the goal configuration respectively.
- The trees are expanded not only towards randomly sampled configurations, but also towards the nearest node of the opposite tree with a probability p. Hence, with a probability p the closest distance among the $m \times n$ distances between nodes from both trees is found and the node of the expanding tree is expanded towards the node of the non-expanding tree. With a probability $1 - p$ a random configuration is sampled and the corresponding trees are expanded as usual.
- The *Expand* phase is repeated several times until an obstacle is found. The resulting nodes from the local searches limited by ε are added to the tree. This is called the *Connect* phase.

Growing the trees from the initial and the goal configurations and, at times, towards the opposite tree gives the algorithm its characteristic single-query behavior. The *Connect* phase was added after empirically testing that the additional greediness that it introduced improved the performance in many cases. A common variation is also trying to extend the tree towards the opposite tree after every q_{new} is added when sampling randomly, extending from that q_{new} configuration towards the opposite tree. This helps in cases in which both trees are stuck in regions of the search space that are close as per the distance measure, but in which local searches consistently fail due to obstacles.

RRT-Plan

The planner RRT-Plan (Burfoot, Pineau, and Dudek 2006) was proposed as a stochastic planning algorithm inspired by RRTs. In this case, the EHC search phase of FF (Hoffmann and Nebel 2001), a deterministic propositional planner, was used as the local planner. The limit ε that was used to limit the reach of the *Expand* phase was substituted by a limit on the number of nodes expanded by the local planner. In this case, once the limit ε was reached the node in the local search with the best heuristic estimate towards the sampled space was chosen, and that node was added to the tree. The tree was built only from the initial state due to the difficulty of performing regression in automated planning.

The key aspects in this work are two: the computation of the distance necessary to find the nearest node to the sampled or the goal state, and sampling in an implicit search space. In RRTs one of the most critical points is the computation of the nearest node in every *Expand* step, which may become prohibitively expensive as the size of the tree grows with the search. The most frequently used distance estimations in automated planning are the heuristics based on the reachability analysis in the relaxed problem employed by forward search planners, like the h^{add} heuristic used by HSP (Bonet and Geffner 2001) or the relaxed plan heuristic introduced by FF (Hoffmann and Nebel 2001). The problem with these heuristics is that, although computable in polynomial time, they are usually still relatively expensive to compute. To avoid recomputing the reachability analysis from every node in the tree, every time a new local search towards a state is done, the authors propose caching the cost of achieving every goal proposition from a node whenever that node is added to the tree. This way, by adding the costs of the propositions that form the sampled state, h^{add} can be obtained without needing to perform a reachability analysis.

Regarding sampling, RRT-Plan does not sample the search state uniformly. Instead, it chooses a subset of propositions $s \subseteq S$ from the goal set such that $s \subseteq G$ and uses s as q_{rand}. This is due to the fact that, although sampling a state by choosing random propositions in automated planning is trivial, determining whether a given sampled state belongs to the search space is PSPACE-complete, as it is as hard, in terms of computational complexity, as solving the original problem itself. This problem is avoided by the sampling technique of RRT-Plan in the sense that, if the problem is solvable, G must be reachable. Hence, any of its possible subsets is also reachable. In addition, RRT-Plan performs goal locking; i.e., when a goal proposition p that was part of a given sampled state $s \subseteq G \mid p \in s$ is achieved, any subsequent searches from the added q_{new} node and its children nodes are not allowed to delete p.

Whereas RRT-Plan effectively addresses the problem of sampling states in implicit search spaces, this kind of sampling limits most of the advantages RRTs have to offer. By choosing subsets of the goal set instead of sampling uniformly the search space, the RRT does not tend to expand towards unexplored regions. Thus, it loses the implicit balance between exploration and exploitation during search that characterizes them. In fact, by choosing this method, RRT-Plan actually benefits from random guesses over the order of the goals instead of exploiting the characteristics of RRTs. As a side note, this could actually be seen as a method similar to the goal agenda (Koehler and Hoffmann 2000), albeit with random selection of subsets and the possibility in this case to recover from wrong orderings.

Advantages of RRTs in Automated Planning

Figure 4 shows a typical example of a best-first search algorithm getting stuck in an *h plateau* due to inaccuracies in the heuristic. In this example, the euclidean distance used as heuristic ignores the obstacles. Because of this, the search advances forward until the obstacle is found. Hence, the search algorithm must explore the whole striped area before it can continue advancing towards the goal. This highlights the imbalance between exploitation and exploration these approaches have. This problem has been previously studied, and several methods that tried to minimize its negative impact on search have been proposed (Röger and Helmert 2010; Linares López and Borrajo 2010). However, this imbalance still remains as one of the main shortcomings of best-first search algorithms. To partially address this issue, we consider expanding nodes towards randomly sampled states so a more diverse exploration of the search space is done. In this example, a bias that would make the search advance towards q_{rand} could avoid the basin flooding phenomenon that greedier approaches suffer from.

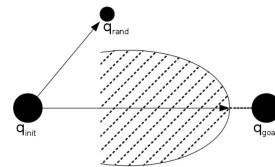


Figure 4: Simple example of a best-first search algorithm greedily exploring an *h plateau* due to the heuristic ignoring the obstacles. Advancing towards some randomly sampled state like q_{rand} can alleviate this problem.

RRTs incrementally grow a tree towards both randomly sampled states and the goal. Therefore, they are less likely to suffer from the same problem as best-first search algorithms. The main advantages that they have over other algorithms in automated planning are the following:

- They keep a better balance between exploration and exploitation during search.
- Local searches minimize exploring plateaus, as the maximum size of the search is bounded.
- They use considerably less memory, as only a relatively sparse tree must be kept in memory.
- They can employ a broad range of techniques during local searches.

In terms of memory, the worst case is the same for best-first search algorithms and RRTs. However, RRTs must keep in memory only the tree and the nodes from the current local search. Trees are typically much sparser than the area explored by best-first algorithms, which makes them much more memory efficient on average.

Implementing RPT

Due to the differences in the search space, adapting RRTs from motion planning to automated planning is not trivial. RPT is an planner based on RRTs and RRT-Plan with some changes critical to their performance.

Sampling

The main reason why RRTs have not been considered for automated planning is the difficulty of properly sampling the search space. The difficulty arises from the fact that choosing propositions from S at random may lead to generating spurious states. Checking whether a state is spurious or not is as hard as solving the problem itself, so an approximative approach must be used instead. Here we propose the use of state invariants as constraints to reduce the chances of obtaining a spurious state when uniformly sampling the search space. In particular, we propose the use of mutexes (Bonet and Geffner 2001) (already employed by evolutionary planners like $D_A E_X$ (Bibai et al. 2010), which decomposes the problem using sampling techniques) and *"exactly-1"* invariant groups.

Sampling a state using state invariants as constraints is analogous to solving a Constraint Satisfaction Problem (CSP). A CSP is formally defined as a triple $CSP=(V,D,C)$, where V are the variables of the problem, D are the domains of the variables in D and C are the constraints of the problem. In this CSP the *"exactly-1"* invariant groups are the variables in V, the propositions of the *"exactly-1"* invariant groups are the domain D of the variables in V and the binary mutexes of the problem are the constraints of C. The objective is to choose a proposition $p \in S$ from every *"exactly-1"* invariant group I_1 such that it is not mutex with any other chosen proposition $p' \in S$. This ensures that the complete sampled state $s \in S$ satisfies all *"exactly-1"* invariant groups and does not violate any binary mutex.

Solving a CSP is NP-complete. Actually, for some planning instances solving the CSP that represents the sampling process may be on average very time consuming if it is done naively. In our implementation we use forward checking (Haralick and Elliott 1980) to improve the performance of the backtracking procedure needed for solving the CSP. The order of the variables (the order in which the *"exactly-1"* invariant groups are selected to be satisfied) is static, although it may vary between different sampling processes. *"exactly-1"* invariant groups with the highest cardinality are chosen first, with ties broken randomly every time a new state is sampled. Ordering of values of variables is chosen at random. This aims to reproduce the behavior of the degree (most constraining variable) heuristic (Brélaz 1979) while trying to obtain sampled states as diverse as possible.

Ensuring the Reachability of Goals Even after using state invariants, it may happen that the goal is not reachable from the sampled state. For example, a sampled state in the *Sokoban* domain may contain a configuration of blocks such that some block cannot be moved anymore. While this sampled state may not violate any state invariant, unless the unmovable blocks are at a goal location the sampled state is a dead end, since the original goal is not reachable. To address this problem, a regular reachability analysis can be done from the sampled state. If some proposition $p \in G$ is unreachable, then the sampled state can be safely discarded. This is again an incomplete method, but in cases such as the aforementioned one it is useful to detect spurious states.

Distance Estimation

One of the most expensive steps in an RRT is finding the closest node to a sampled state. Besides, the usual distance estimation in automated planning, the heuristics derived from a reachability analysis, are also computationally costly. RRT-Plan solved this by caching the cost of achieving a goal proposition from every node of the tree and using that information to compute h^{add}, just like HSPr does (Bonet and Geffner 2001) when searching backwards. Despite being an efficient solution, this shares the same problem as HSPr: only h^{add} can be computed using that information. h^{add} tends to greatly overestimate the cost of achieving the goal set and other heuristics of the same kind, like the FF heuristic, are on average more accurate (Fuentetaja, Borrajo, and Linares López 2009). Therefore, in our implementation, *best supporters*, that is, actions that first achieve a given proposition in the reachability analysis, are cached as proposed by Alcázar *et al* (Alcázar et al. 2013). This allows to compute not only h^{add} but also other heuristics like the FF heuristic (by tracing back the relaxed plan using the cached best supporters). The time of computing the heuristic once the best supporters are known is usually very small compared to the time needed to perform the reachability analysis - linear in the size of the relaxed plan -, so this approach allows to get more accurate (or diverse) heuristic estimates without incurring in a significant overhead.

Tree Construction

RRTs can be built in several ways. The combination of the *Extend* and *Connect* phases, the possibility of greedily advancing towards the goal with a probability $1-p$ instead of sampling with a probability p, the way new nodes are added (only the closest node to the sampled state or all the nodes on the path to that state),... allow for a broad range of different options. In this work, we have chosen to build the tree in the following way:

- the tree is built from the initial state I;
- every node in the tree contains a state, a link to its parent, a plan that leads from its parent to the state, and the cached best supporters for every proposition $q \in S$ so h^{FF} can be computed efficiently;
- ε limits the number of expanded nodes in every local search;
- there is a probability p of advancing towards a sampled state and a probability $1-p$ of advancing towards the goal from the closest node to the original goal G. It may happen that the closest node to G was already expanded towards G in an earlier iteration and the new generated node q_{new} from that expansion is farther from the goal than G; that is, $h^{FF}(q_{new}) > h^{FF}(q_{near})$. Since planners are for the most part deterministic, it does not make sense to repeat

the search - it would lead to the same q_{new} -, so in fact the node selected with a probability $1-p$ is the closest node *among those that have never served as the origin of a local search towards the goal before*.

- a single node is added to the tree after every local search, not all the nodes along the solution path;
- when performing a local search, if a solution for the subproblem was not found, the last expanded node is added to the tree (be it when expanding towards a sampled state or the original goal G itself);
- after adding a new node q_{new} from the local search towards a sampled state, a new local search from q_{new} to q_{goal} is performed.

No *Connect* phase is performed. This is because the *Connect* phase is probably counter-productive if it is done towards sampled states - sampled states may be completely irrelevant to the solution and the main benefit obtained from them is the additional bias towards exploration anyway - and partially overlaps with the expansions towards the goal with a probability $1-p$. Algorithm 2 describes the whole process.

Algorithm 2: Search process of RPT.

```
Data: Search space S, limit ε, initial state q_new, goal
      q_goal
Result: Plan solution
begin
    tree ⟵ q_init
    while ¬ goalReached() do
        if p <random() then
            q_rand ⟵ sampleSpace(S)
            q_near ⟵ nearest(tree,q_rand,S)
            q_new ⟵ join(q_near,q_rand,ε,S)
            addNode(tree,q_near,q_new)
            q_near_goal ⟵ q_new
        else
            q_near_goal ⟵ nearest(tree,q_goal,S)
        q_new_goal ⟵ join(q_near_goal,q_goal,ε,S)
        addNode(tree,q_near_goal,q_new_goal)
    solution ⟵ traceBack(tree,q_goal)
    return solution
end
```

Choice of the Local Planner

The choice of the planner used in the local search is subject to some restrictions. First, after every *Extend* phase a new node to the tree is added even if a solution for the subproblem could not be found. This means that the local planner must be able to return an executable plan also when no solution was found, which rules out some planning paradigms like partial-order planners (Younes and Simmons 2003) and SAT-based planners (Rintanen 2012). Second, the tree is built forward, so the local planner must return a forward-executable plan. Again, backward search planners like HSPr (Bonet and Geffner 2001) and FDr (Alcázar et al. 2013) cannot be used for this reason. Another important point is the preprocessing time. Since multiple local searches may be done, it is desirable that the time spent by the local planner prior to search is as short as possible. For example, the use of heuristics that require a relatively long preprocessing time and depend on either the initial state or the goals, like Pattern Databases (Culberson and Schaeffer 1998), are discouraged.

In this work, the Fast Downward planning system (Helmert 2006) was used as the local planner. It was configured to use greedy best-first search with lazy evaluation as its search algorithm. The heuristic is the FF heuristic (Hoffmann and Nebel 2001). Preferred operators obtained from the FF heuristic were enabled.

Implementation Details and Parameters

RPT was implemented on top of Fast Downward (Helmert 2006). Since RRTs are stochastic algorithms, the seed is fixed for the pseudorandom number generator so results would be reproducible. In particular, we use the default seed '1' for the *rand* function of the standard C++ library. The computation of h^2 was implemented in Fast Downward; mutexes were obtained from the computation of h^2 forward and backward. *"exactly-1"* invariant groups were obtained from the monotonicity analysis done by the translator of Fast Downward. To further exploit the state invariants, spurious actions were pruned by disambiguating their preconditions (Alcázar et al. 2013). We set a limit of 300 seconds for the h^2 computation and the disambiguation of actions. ε was set to $\varepsilon = 50000$; p was set to $p = 0.5$.

Anytime Phase

Since quality matters in the competition, we have enabled an anytime phase that begins right after the first solution is found. It searches forward iteratively using the following configurations: Greedy Best First Search with delayed evaluation, a cost-sensitive version of the FF heuristic and preferred operators; Weighted $A^\star$ with regular evaluation, a cost-sensitive version of the FF heuristic, preferred operators and $w = 5, 3, 1$ sequentially; $A^\star$ with a cost-sensitive version of the FF heuristic and no preferred operators; and blind search.

These settings haven't been thoroughly tested, they were mainly just a combination of those of LAMA and our intuition.

Acknowledgements

This work has been partially supported by a FPI grant from the Spanish government associated to the MICINN project TIN2008-06701-C03-03, and it has also been supported by the project TIN2011-27652-C03-02.

The fourth author was partially funded by the Office of Naval Research under grant number N00014-09-1-1031. The views and conclusions contained in this document are those of the authors only.

References

Alcázar, V.; Borrajo, D.; Fernández, S.; and Fuentetaja, R. 2013. Revisiting regression in planning. In *International Joint Conference on Artificial Intelligence*, 2254–2260.

Alcázar, V.; Veloso, M. M.; and Borrajo, D. 2011. Adapting a Rapidly-Exploring Random Tree for automated planning. In *Symposium on Combinatorial Search*, 2–9.

Aurenhammer, F. 1991. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys* 23(3):345–405.

Bibai, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010. An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning. In *International Conference on Automated Planning and Scheduling*, 18–25. AAAI.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Brélaz, D. 1979. New methods to color the vertices of a graph. *Communications of the ACM* 22(4):251–256.

Burfoot, D.; Pineau, J.; and Dudek, G. 2006. RRT-Plan: A randomized algorithm for STRIPS planning. In *International Conference on Automated Planning and Scheduling*, 362–365.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Comput. Intell.* 14(3):318–334.

Fuentetaja, R.; Borrajo, D.; and Linares López, C. 2009. A unified view of cost-based heuristics. In *Proceedings of the "2nd Workshop on Heuristics for Domain-independent Planning". Conference on Automated Planning and Scheduling (ICAPS'09)*.

Haralick, R. M., and Elliott, G. L. 1980. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.* 14(3):263–313.

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

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

Koehler, J., and Hoffmann, J. 2000. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *J. Artif. Intell. Res. (JAIR)* 12:338–386.

Kuffner, J. J., and LaValle, S. M. 2000. RRT-Connect: An efficient approach to single-query path planning. In *ICRA*, 995–1001. IEEE.

LaValle, S. M., and Kuffner, J. J. 1999. Randomized kinodynamic planning. In *International Conference on Robotics and Automation*, 473–479.

Linares López, C., and Borrajo, D. 2010. Adding diversity to classical heuristic planning. In *Proceedings of the Third Annual Symposium on Combinatorial Search (SOCS'10)*, 73–80.

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

Röger, G., and Helmert, M. 2010. The more, the merrier: Combining heuristic estimators for satisficing planning. In *International Conference on Automated Planning and Scheduling*, 246–249.

Younes, H. L. S., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *J. Artif. Intell. Res. (JAIR)* 20:405–430.

USE: The Useful Operator Selection

Reza Sadraei
University of Sharif, Tehran, Iran
sadraei_reza@yahoo.com

Atefeh Ahmadi
University of Science and Technology, Tehran, Iran
atefeh_31@yahoo.com

Abstract

USE is a sequential portfolio planner that uses two search algorithms: forward and backward search. USE is implemented based on the Fast Downward planning system. The forward search employed in USE is a stochastic planner that uses "Useful Operator Selection" in addition to heuristic functions and preferred operators of LAMA2011. "Useful Operator Selection" policy is stochastic and sometimes causes the goal to be unreachable from the initial state. In such a case, search process does not progress after a while and USE resets the forward search process, hoping to select a better operator in the next iteration. USE repeats this procedure till the search process progresses continuously and the goal is reached. In a few domains the backward search could solve the planning problems better than the forward search; Consequently, USE also applies this mechanism. In the backward search, USE utilizes additive heuristic without preferred operator.

Introduction

Recently, many of heuristic planners use preferred operators to limit their search spaces. These planners can search in a smaller space and reach the goals faster. In most of these planners, selection of preferred operators is done by using heuristic functions. The performance of employing preferred operators in planning domains is related to the performance of heuristic functions in that domain. It means that, if the heuristic function can be useful for a specific domain, the selection of preferred operators will be helpful. On the other hand, in domains that heuristics are not accurate, preferred operators cannot usually lead the search process to the goals. In this case, preferred operators lose their power in leading search toward the goal, and the planner must search a big part of the state space for reaching the goals. For example, the Fast Forward Planner uses Relaxed Planning Graph (Hoffmann and Nebel 2001) to compute the heuristic function and also employs this structure to select the helpful actions. The Relaxed Planning Graph is built by ignoring the negative effects of actions. If this relaxation changes the nature of the task, both heuristic function and helpful actions can't guide the search efficiently.

In addition to preferred operators, USE applies another mechanism to prune the search space. This mechanism, called "Useful Operator Selection", is totally independent of heuristic functions and is based on the nature of the planning tasks. In cases that heuristic functions and preferred operators can't guide the search efficiently, applying the "Useful Operator Selection" prunes the search space and USE can find a valid plan faster.

On the other hand, the "Useful Operator Selection" mechanism isn't complete; it means that, this method may prune parts of the state space that include the goal. Since this pruning mechanism is stochastic, when the search does not progress, USE will reset the search process. In the cases that the search process sticks in the plateau or the goal states are pruned, there will be no progress in the search. In these cases, the resetting of the process can be useful: it may rescue the search from the plateau or the next search iteration may not prune the goal.

Useful Operators

Because of the large scale of the search spaces in planning problems, many planners use some methods like heuristic search and employing preferred operators for efficient search in these problems. In addition to the mentioned methods, we have applied another mechanism in USE to make the search space smaller. In this mechanism, which is similar to using preferred operators, some of the applicable operators for the current state are selected and other operators are ignored. We call the selected operators Useful Operators.

Suppose that two operators are applicable to current state and the order of applying them is not important. In other words, the order of applying them does not effect on the result and the same state will be generated after

applying both of them. These two operators must not interfere with each other, and none of them must produce precondition for the other. These operators are called independent operators. When exploring the search space, only one of these operators is needed to be applied to the current state. The other operator can be ignored because it can be applied to the next state. In these cases the randomly selected operator is called Useful Operator.

Two operators are independent if in each valid plan they can apply in parallel and the order of executing them does not have any effect on other operators. This concept is the opposite of the Mutual Exclusive Relation in Planning Graphs (Avrim and Blum 1997). Two actions at a given action level in the Planning Graph are mutually exclusive if no valid plan could possibly include both. It means that, if two operators are exclusive, they will not be independent. If two operators are Independent and they are applicable to the current state, instead of generating two new states by applying both of them, USE selects one of them randomly and applies that operator to the current state. USE utilizes the concept of mutual exclusive relation to distinguish independent operators. In other words, if two operators do not be exclusive, they will be independent. Distinguishing independent operators are costly, so the same method in (Avrim and Blum 1997) is used. Two operators are Exclusive if they have one of the following conditions:

Interference: If either of the actions deletes a precondition or Add-Effect of the other.

Competing Needs: If there is a precondition of action “A” and a precondition of action “B” that are marked as mutually exclusive of each other in the previous proposition level.

It should be clear that the competing needs between two operators that is applicable to current state, never happens. As long as this rule does not distinguish all exclusive operators, two operators may be considered independent, while the order of applying them is important. In this case, some regions of the search space are lost while some goal states may be in those regions. So, this pruning method is not complete and USE uses this method in an iterative method.

When more than two operators could be applied to current state, the following algorithm is used to find the Useful Operators:

```
UsefulOps ← φ
for each op in ApplicabaleOps then
   if for all op1 ∈ UsefulOps, op and op1 are
   exclusive then
      UsefulOps ← UsefulOps ∪ {op}
```

Iterative Search Process

As mentioned in the past section, the “Useful Operator Selection” strategy is not complete. Therefore, it is used in an iterative search approach. In this approach when the search does not progress, it will be reset by clearing open lists and close lists. A simple mechanism for resetting the search process is applied in USE. Assume that, planner reaches the state with the best heuristic so far, when the close list size is equal to “n”. One method for determining the moment of resetting is comparing “n” with the close list size. It means that, when the close list size equals to “Kn” (K is configurable parameter) and the planner does not reach a state with better heuristic value, the search process will be reset. In some domains, such as “Nomystry”, a lot of states are dead-ends which causes the search process to be reset rapidly. In these domains, the search process does not have enough time for searching neighbor state space. To overcome this problem, USE only considers the states that are not dead-ends. So resetting the search process occurs after extracting “Kn” states from open list that are not dead-end.

In one domain in “IPC 2011”, the proportion of pruned operator in all applicable operators is negligible. In such domains, after resetting the search, the next iteration goes thorough the previous iterations footsteps. So USE evaluates this ratio and resets the search when it is bigger than ϵ, where ϵ is a small number and is configurable.

Backward Search

Another approach that is implemented in USE is the backward search. In this approach, each node of the search space is a partial state. In the partial states some state variables don’t have a value. The goal is a partial state and it is considered as initial node in the backward search. In the backward search, actions are applied reversely to partial states. An action will be reversely applicable to a partial state if each variable in that partial state has one of the following conditions:

- The variable does not appear in the preconditions or effects of the operator.
- The variable has the same value in the partial state and the operator’s effects.
- The variable has the same value in partial state and operator’s precondition, and the value of that variable is not changed by the operator’s effects.

The result of reversely applying an operator to a partial state is a new partial state as:

- The variables of parent partial state that don’t appear in operator’s preconditions have the same value in child partial state.

- All state variables that appear in operator's precondition are added to the child partial state with their corresponding values in operator's precondition.

Producing unreachable partial states is one of the main drawbacks of backward search. The state variable representation decreases the number of unreachable state generation in backward search compared to the classical representation. In addition, during its translation phase, LAMA2011 generate a file that contains groups of mutually inconsistent predicates. By using these groups, USE avoid from generating partial states that contain inconsistent predicates. By this method branching factor decreases drastically and therefore, the search space will be considerably pruned. The implemented backward search only contains the additive heuristic functions, and not preferred operators.

For most of the domains in "IPC 2011", this approach cannot work efficiently, whereas in some other domains backward search has an enormous effect on increasing the performance of the search. To test this approach in "IPC 2014", USE is using the backward search in a portfolio structure with the forward search method, where only a small time slice is allocated to the backward search method.

References

Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. Articial Intelligence 90:279–298, 1997.

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

Silvia Richter and MatthiasWestphal. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. JAIR, 39:127–177, 2010.

YAHSP3 and YAHSP3-MT in the 8th International Planning Competition

Vincent Vidal
Onera - The French Aerospace Lab
Toulouse, France
Vincent.Vidal@onera.fr

Description

YAHSP3 (Vidal 2004) is a forward state-space heuristic search planner that embeds a lookahead policy based on an analysis of relaxed plans. The core of the solver has nearly not evolved since IPC-2011 where YAHSP2 competed, and is described in full details in (Vidal 2011). It can be noted that a minor bug with major effects has been fixed, which prevented YAHSP2 to find valid plans in domains with O-cost actions (YAHSP2 got a score of 0 in all such domains at IPC-2011). The multi-threaded version YAHSP3-MT is also nearly identical to YAHSP2-MT, and is described in (Vidal, Bordeaux, and Hamadi 2010).

YAHSP{2,3} have been used in different projects:

- **Parallel planning on distributed memory machines.** YAHSP has been parallelized following the ideas of HDA* (Kishimoto, Fukunaga, and Botea 2009) with the MPI library and evaluated on two kinds of machines with a distributed memory architecture: a small-sized cluster consisting of 4 servers with 12 cores each, and an experimental many-core processor developed by Intel Labs, the Single-chip Cloud Computer (SCC), containing 48 cores on a mesh. Super-linear speedups are often observed, particularly on the SCC thanks to the efficiency of its internal network (Vidal, Vernhes, and Infantes 2011).
- **The Landmark-based Meta Best-First Search algorithm (LMBFS).** The objective was to perform a meta-search in the space of landmark orderings, in order to find a sequence of landmarks that could help an underlying planner to find a solution (Vernhes, Infantes, and Vidal 2012; 2013b). A parallelization of the meta-search algorithm inspired by (Vidal, Vernhes, and Infantes 2011) has been proposed in (Vernhes, Infantes, and Vidal 2013a), but has not produced interesting results yet.
- **Multi-objective AI planning.** The DaE planner (Schoenauer, Savéant, and Vidal 2006; 2008; Bibaï et al. 2010) that embeds YAHSP has been extended with multi-objective evolutionary algorithms (NSGA-II, SPEA2, IBEA_{H}) in order to generate Pareto fronts, and studied following different perspectives (Khouadjia et al. 2013b; 2013d; 2013a; 2013c). Experimental results have been produced on modified benchmarks from the IPC for supporting several objectives.

References

Bibaï, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2010. An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-2010)*, 18–25. Toronto, ON, Canada: AAAI Press.

Khouadjia, M.-R.; Schoenauer, M.; Vidal, V.; Dréo, J.; and Savéant, P. 2013a. Multi-objective AI planning: Comparing aggregation and pareto approaches. In *Proceedings of the 13th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP-2013)*, volume 7832 of *LNCS*, 202–213. Vienna, Austria: Springer.

Khouadjia, M.-R.; Schoenauer, M.; Vidal, V.; Dréo, J.; and Savéant, P. 2013b. Multi-objective AI planning: Evaluating DAEyahsp on a tunable benchmark. In *Proceedings of the 7th International Conference on Evolutionary Multi-Criterion Optimization (EMO-2013)*, volume 7811 of *LNCS*, 36–50. Sheffield, UK: Springer.

Khouadjia, M.-R.; Schoenauer, M.; Vidal, V.; Dréo, J.; and Savéant, P. 2013c. Pareto-based multiobjective ai planning. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-2013)*. Beijing, China: AAAI Press.

Khouadjia, M.-R.; Schoenauer, M.; Vidal, V.; Dréo, J.; and Savéant, P. 2013d. Quality measures of parameter tuning for aggregated multi-objective temporal planning. In *Proceedings of the 7th Learning and Intelligent Optimization Conference (LION-2013)*, LNCS. Catania, Italy: Springer.

Kishimoto, A.; Fukunaga, A. S.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *Proceedings of the 5th International Planning Competition (IPC-2011)*, 10–17.

Schoenauer, M.; Savéant, P.; and Vidal, V. 2006. Divide-and-Evolve: a new memetic scheme for domain-independent temporal planning. In *Proceedings of the 6th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP-2006)*, volume 3906 of *LNCS*, 247–260. Budapest, Hungary: Springer.

Schoenauer, M.; Savéant, P.; and Vidal, V. 2008. Divide-and-Evolve: a sequential hybridization strategy using evolutionary algorithms. In Michalewicz, Z., and Siarry, P., eds.,

Advances in Metaheuristics for Hard Optimization, Natural Computing Series. Springer. chapter 9, 179–198.

Vernhes, S.; Infantes, G.; and Vidal, V. 2012. The landmark-based meta best-first search algorithm for classical planning. In *Proceedings of the 5th European Starting AI Researcher Symposium (STAIRS-2012)*, volume 241 of *Frontiers in Artificial Intelligence and Applications*, 336–347. Montpellier, France: IOS Press.

Vernhes, S.; Infantes, G.; and Vidal, V. 2013a. Landmark-based meta best-first search algorithm: First parallelization attempt and evaluation. In *Proceedings of the 10th ICAPS Workshop on Heuristics and Search for Domain-independent Planning (HSDIP-2013)*.

Vernhes, S.; Infantes, G.; and Vidal, V. 2013b. Problem splitting using heuristic search in landmark orderings. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-2013)*. Beijing, China: AAAI Press.

Vidal, V.; Bordeaux, L.; and Hamadi, Y. 2010. Adaptive K-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning. In *Proceedings of the 3rd Symposium on Combinatorial Search (SOCS-2010)*, 100–107. Stone Mountain, GA, USA: AAAI Press.

Vidal, V.; Vernhes, S.; and Infantes, G. 2011. Parallel AI planning on the SCC. In *Proceedings of the 4th Symposium of the Many-core Applications Research Community (MARC-2011)*, 15–20. Potsdam, Germany: Hasso-Plattner-Institute Press.

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-2004)*, 150–159. Whistler, BC, Canada: AAAI Press.

Vidal, V. 2011. YAHSP2: Keep it simple, stupid. In *Proceedings of the 7th International Planning Competition (IPC-2011)*, 83–90.

Madagascar: Scalable Planning with SAT

Jussi Rintanen[*]
Department of Information and Computer Science
Aalto University
Helsinki, Finland

Abstract

The scalability of SAT to very large problems has been achieved in the last ten years, due to substantially improved encodings, SAT solvers, and algorithms for scheduling the runs of SAT solvers. This has lead to SAT-based planners that are dramatically different from the earliest implementations based on the late 1990ies technology. We discuss a SAT-based planning system that implements modernized versions of all components of earliest SAT-based planners.

Introduction

During the last decade, SAT, the prototypical NP-complete problem of testing the satisfiability of the formulas in the classical propositional logic (Cook 1971), has emerged, due to dramatically improved SAT solvers (Marques-Silva and Sakallah 1996; Moskewicz et al. 2001) as a *practical* language for representing hard combinatorial search problems and solving them, in areas as diverse as Model-Checking (Biere et al. 1999), FPGA routing (Wood and Rutenbar 1998), test pattern generation (Larrabee 1992), and diagnosis (Smith et al. 2005; Grastien et al. 2007).

Planning as Satisfiability, which enjoyed a lot of attention in the late 1990s after the works by Kautz and Selman (1996), has re-emerged as a strong approach to planning due to substantially improved problem encodings, SAT solvers, and search strategies. The main application is the classical planning problem (Rintanen 2012b), but the same ideas can be adapted to more complex forms of planning, or classical planning can be used as a subprocedure in algorithms for more general problems. These investigations have only started, with first breakthroughs obtained in temporal planning (Rankooh and Ghassem-Sani 2013).

These developments are not surprising, considering that the classical planning problem is equivalent to the simplest model-checking and reachability problems in Computer-Aided Verification, and that SAT and its extensions such as SAT modulo Theories (SMT) have had great successes in that area in the past ten years, including wide industry adoption.

[*]Also affiliated with Griffith University, Brisbane, Australia, and Helsinki Institute of Information Technology, Finland. This work was funded by the Academy of Finland (Finnish Centre of Excellence in Computational Inference Research COIN, 251170).

In this paper, we will first give a brief description of the Planning and Satisfiability approach, discuss the issues critical to its time and space complexity in practice, and explain the factors that separate the state of the art now and ten years ago. Specifically, we discuss two critical issues of SAT-based planning: potentially high memory requirements, and the necessity and utility of guaranteeing that plans have the shortest possible horizon length (parallel optimality).

The planning system Madagascar (also called M, Mp or MpC depending on its configuration) implements several of the innovations in planning with SAT, including compact and efficient encodings based on $\exists$-step plans (Rintanen, Heljanko, and Niemelä 2006), parallelized/interleaved search strategies (Rintanen 2004; Rintanen, Heljanko, and Niemelä 2006), powerful invariant algorithms (Rintanen 2008b), SAT heuristics specialized for planning (Rintanen 2010b; 2010a), and data structures supporting parallelized SAT solving with very large problem instances (Rintanen 2012a).

Background

A classical planning problem is defined by a set F of facts (or state variables) the valuations of which correspond to states, one initial state, a set A of actions (that represent the different possibilities of changing the current state), and a goal which expresses the possible goal states in terms of the facts F. A solution to the planning problem is a sequence of actions that transform the initial state step by step to one of the goal states.

The classical planning problem can be translated into a SAT problem of the following form.

$$\Phi_t = I \wedge \mathcal{T}(0,1) \wedge \mathcal{T}(1,2) \wedge \cdots \wedge \mathcal{T}(t-1,t) \wedge G$$

Here I represents the unique initial state, expressed in terms of propositional variables $f@0$ where $f \in F$ is a fact, and G represents the goal states, expressed in terms of propositional variables $f@t, f \in F$. The formulas $\mathcal{T}(i, i+1)$ represent the possibilities of taking actions between time points i and $i+1$. These formulas are expressed in terms of propositional variables $f@i$ and $f@(i+1)$ for $f \in F$ and $a@i$ for actions $a \in A$.

The formula Φ_t is satisfiable if and only if a plan with t time points exists. Planning therefore can be reduced to

a sequence of satisfiability tests. The effectiveness of the planner based on this idea is determined by the following.

1. The form of the formulas $\mathcal{T}(i, i+1)$.
2. The way the values of t are chosen.
3. The way the SAT instances Φ_t are solved.

In the rest of the paper we will discuss each of these components of an efficient and scalable planning system that uses SAT.

Encodings of $\mathcal{T}(i, i+1)$

The encoding of transitions from i to $i+1$ as the formulas $\mathcal{T}(i, i+1)$ determines how effectively the satisfiability tests of the formulae Φ_t can be performed. The leading encodings are the factored encoding of Robinson et al. (2009), and the $\exists$-step encoding of Rintanen et al. (2006). Both of them use the notion of *parallel* plans, which allow several actions at each time point and hence time horizons much shorter than the number of actions in a plan. The encoding by Robinson et al. is often more compact than that by Rintanen et al., but the latter allows more actions in parallel. Both of these encodings are often more than an order of magnitude smaller than earlier encodings such as those of Kautz and Selman (1996; 1999), and also substantially more efficient (Rintanen, Heljanko, and Niemelä 2006; Sideris and Dimopoulos 2010). This is due to the very large quadratic representation of action exclusion in early encodings. Rintanen et al. (2006) and Sideris and Dimopoulos (2010) show that eliminating logically redundant mutexes or improving the quadratic representation to linear dramatically reduces the size of the formulas.

The $\exists$-step plans allow more actions in parallel than the earlier most popular GraphPlan-style (Blum and Furst 1997) $\forall$-step plans (Dimopoulos, Nebel, and Koehler 1997; Rintanen, Heljanko, and Niemelä 2006). Further, the weaker conditions on parallelism for $\exists$-step plans often allow leaving out all constraints on the parallelity of actions, which further leads to smaller formulas than with $\forall$-step plans (Rintanen, Heljanko, and Niemelä 2006). Both factors, shorter horizon lengths and smaller encodings for action parallelism, substantially help improving the scalability of SAT-based planning.

In addition to constraints that are necessary for the correctness of planning, there are *redundant* constraints that logically follow from the necessary constraints, but that are still useful because they make such implicit facts explicit that would otherwise not be effectively inferred by the SAT solver (Rintanen 2008a).

Our planners use *invariants* (binary mutexes) to speed up SAT solving. Mutexes were first introduced in the GraphPlan planner (Blum and Furst 1997) for pruning a backward-chaining search, and they were soon noticed to be important also for SAT-based planning (Kautz and Selman 1996). The main reason for the utility of mutexes in planning is the representation of multi-valued state variables as sets of Boolean variables. That multi-valued state variables cannot be represented directly is a limitation of the PDDL language used by many planners.

We use a powerful algorithm for finding 2-literal invariants (Rintanen 2008b). The algorithm uses a fixpoint computation similarly to GraphPlan's planning graph construction, but works for a far more general input language that includes arbitrary disjunctions and conditional effects.

Scheduling the Solution of the SAT Instances

Kautz and Selman (1996) proposed testing the satisfiability of Φ_t for different values of $t = 0, 1, 2, \ldots$ sequentially, until a satisfiable formula is found. This strategy is asymptotically optimal if the t parameter corresponds to the plan quality measure to be minimized, as it would with sequential plan encodings that allow at most one action at a time. However, for the parallel $\exists$-step and $\forall$-step plans optimality of the t parameter is meaningless because the parallelism does not correspond to the actual physical possibility of taking actions is parallel. For STRIPS, Graphplan-style parallelism exactly matches the possibility of totally ordering the actions to a sequential plan (Rintanen, Heljanko, and Niemelä 2006). Hence the parallelism can be viewed as a form of partial order reduction (Godefroid 1991), the purpose of which is to avoid considering all $n!$ different ordering of n independent actions, as a way of reducing the state-space explosion problem. In this context the t parameter often only provides a weak lower bound on the sequential plan length. So if the minimality of t does not have a practical meaning, why minimize it? The proof that t is minimal was the most expensive part of the runs of early SAT-based planners.

More complex algorithms for scheduling the SAT tests for different t have been proposed and shown both theoretically and in practice to lead to dramatically more efficient planning, often by several orders of magnitude (Rintanen 2004; Zarpas 2004; Streeter and Smith 2007). These algorithms avoid the expensive proofs of minimality of the parallel plan length, and in practice still lead to plans of comparable quality to those with the minimal parallel length. The most effective implementations of these algorithms solve several SAT problems (for different horizon lengths) in parallel.

Algorithm B (Rintanen 2004) runs an unlimited number of SAT solvers at varying rates, solving an sequence of SAT problems for formula $\Phi_0, \Phi_1, \Phi_2, \ldots$. Each SAT solver gets a fraction of the CPU that is proportional to γ^i, for some constant γ that satisfies $0 < \gamma \leq 1$ (we have very successfully used $\gamma = 0.9$). Hence each SAT test for Φ_i gets γ times the CPU the test for Φ_{i-1} gets. Most of the CPU is dedicated to short horizon lengths, but also longer horizon lengths get some CPU. In real-world implementations of the algorithm SAT solvers are started only for Φ_i for which an amount of CPU time is allocated that exceeds some positive threshold value. Our planners in their default configuration also limit the maximum number of SAT instances solved concurrently to 20, with new solvers started when earlier instances are found unsatisfiable.

Figure 1 depicts the gap between the longest horizon length with a completed unsatisfiability test and the horizon length for the found plan for the Mp planner and for all the instances considered by Rintanen (2010b). The dots concentrate in the area below 50 steps, but outside this area there are typically an area of 30 to 50 horizon lengths for

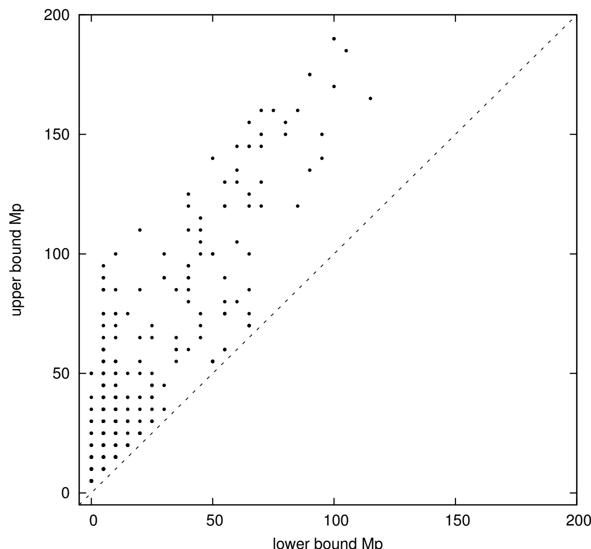


Figure 1: Lower and upper bounds of plan lengths

which the SAT test was not completed, in the vast majority of cases because their difficulty well exceeded the capabilities of current SAT solvers. This explains why the use of the parallel strategies which avoid the expensive (but unnecessary) parallel optimality proofs are essential for efficient planning.

SAT Solving

Our planners are based on our own highly optimized SAT solver that implements the Conflict-Driven Clause Learning algorithm (Marques-Silva and Sakallah 1996; Moskewicz et al. 2001), together with many improvements more or less universally employed in best general-purpose SAT solvers, including phase-saving (Pipatsrisawat and Darwiche 2007), Luby-restarts (Huang 2007), and clause deletion based on literal blocking distance (Audemard and Simon 2009).

In addition to the standard VSIDS heuristic (Moskewicz et al. 2001), our SAT solver implements a planning-specific heuristic which in many cases fares much better than VSIDS on the standard benchmark sets (Rintanen 2012b). The heuristic simulates backward-chaining to identify relevant action variables to be used as decision variables, and leverages the current partial assignment maintained by the CDCL algorithm to focus on the most critical relevant actions. However, standard SAT solvers with VSIDS-style heuristics continue to be the strongest method for solving many combinatorially hard planning problems with relatively short horizon lengths (Porco, Machado, and Bonet 2011; Rintanen 2012b).

In addition to a planning-specific decision variable heuristic, our SAT solver employs a specialized representation for binary clauses (Rintanen 2012a) targeting planning and related state-space search applications such as model-checking. In all standard encodings of planning as SAT, the transition relation formula is replicated multiple times, with different time indices. Our SAT solver includes only one copy of all the binary clauses in the transition relation formula, and handles the varying time indices inside the unit propagation algorithm, with very low overhead. The representation often reduces the memory consumption of the planner to half or one third, and due to reduced cache misses also SAT solving runtimes are often reduced substantially (Rintanen 2012a).

planner	heuristic	scheduling strategy
M	VSIDS	B (geometric rates, linear horizons)
Mp	bwd	B (geometric rates, linear horizons)
MpC	bwd	C (constant rates, exponential horizons)

Table 1: Planner configurations

Versions of the Planner

There is no universal best configuration for our planner (similarly to any other planning method, or combinatorial search method in general), and we have introduced three major configurations which differ in terms of the heuristic and the SAT solver scheduling strategy. These configurations are listed in Table 1.

Planner *M* uses the standard VSIDS heuristic, limits search to plan lengths $5i$ for integers $i \geq 1$, and runs the SAT solvers at varying rates according to the geometric strategy B (Rintanen 2004; Rintanen, Heljanko, and Niemelä 2006). In addition to better encodings, the main difference to early planners that used SAT is the geometric B strategy, which can – in the worst case – be slower than the sequential strategy used by Kautz and Selman only by a small constant factor, but may be – and often in practice is – arbitrarily much faster.

Planner *Mp* is like *M* except that it replaces VSIDS with the heuristic based on backward-chaining which fares exceptionally well with standard planning benchmarks (Rintanen 2010b; 2012b), but often fares worse with smaller but combinatorially harder instances.

Planner *MpC* is like *Mp* but it replaces the horizon lengths $5i$ by horizon lengths $5(\sqrt{2})^i$, with all SAT solvers run at the same rate. *Mp* solves few problem instances with plans much longer than 200 steps due to the difficulty of proving inexistence of long plans and limits on the number horizon lengths considered simultaneously. *MpC* considers longer horizon lengths successfully, up to some thousands of steps, but beyond that it is severely limited by the availability of memory. It sometimes performs worse than *Mp* when the horizon lengths are short.

Conclusions

We have discussed a series of developments that have improved the efficiency and scalability of SAT-based planners dramatically since the early planners from the 1990ies and early 2000s, and that are all implemented in the Madagascar planner.

The single most important improvement – in terms of performance and scalability – was the adoption of parallelized

strategies that do not require the (unnecessary) proof of parallel optimality (Rintanen 2004; 2009). This improvement, together with compact encodings of parallel ∃-step plans (Rintanen, Heljanko, and Niemelä 2004), as implemented in the planner *M*, lifts the efficiency and scalability of SAT-based planning close to the level of the best modern planners that use other search paradigms, and clearly past planners prior to about 2004.

Further improvements for standard benchmark problems have been obtained by replacing general-purpose SAT-solver heuristics, such as VSIDS, by planning-specific ones which help focusing on actions that are relevant and which adapt to the current state of the SAT solving process (Rintanen 2012b), and with various smaller improvements. Finally, of course, a substantial difference to planning as SAT has been the steady and at times dramatic improvement of general-purpose SAT solving technology.

Existing techniques not used by our planners include the use of factored problem encodings for parallel plans (Robinson et al. 2009). With many problems these encodings have outperformed best non-factored encodings, but, as far as we know, the impact of factored encodings for example with parallelized SAT solving strategies (which are critical for high-performance planning) has not been investigated. Overall, the differences of encodings (which were the almost exclusive focus in research on planning with SAT for very long) have a smaller impact on the performance of a SAT-based planner than for example the SAT-solver scheduling strategies. Another method that is not currently used by our planners is approximate plan length upper bounds (Rintanen and Gretton 2013), which would help focusing the search on the horizon lengths most likely to yield plans quickly. The existing method sometimes yields practically significant upper bounds, based on SCCs of dependency graphs, but in many cases decomposition to SCCs is too coarse to be useful. The method is promising, but tighter bounds would be needed to have a substantial impact on planner performance and memory usage.

In summary, the current state of the art in Planning as SAT is characterized by developments in at least half a dozen different planner components. In many cases the improvements in the components have been orthogonal (for example, encodings, SAT solving algorithms, and scheduling of SAT solvers). Understanding dependencies between the different components, most notably between the encodings and algorithms for the SAT problem, would allow further progress in SAT-based methods for planning and state-space search problems in general.

References

Audemard, G., and Simon, L. 2009. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 399–404. Morgan Kaufmann Publishers.

Biere, A.; Cimatti, A.; Clarke, E. M.; and Zhu, Y. 1999. Symbolic model checking without BDDs. In Cleaveland, W. R., ed., *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of 5th International Conference, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, 193–207. Springer-Verlag.

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1-2):281–300.

Cook, S. A. 1971. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 151–158.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs. In Steel, S., and Alami, R., eds., *Recent Advances in AI Planning. Fourth European Conference on Planning (ECP'97)*, number 1348 in Lecture Notes in Computer Science, 169–181. Springer-Verlag.

Godefroid, P. 1991. Using partial orders to improve automatic verification methods. In Guldstrand Larsen, K., and Skou, A., eds., *Proceedings of the 2nd International Conference on Computer-Aided Verification (CAV '90), Rutgers, New Jersey, 1990*, number 531 in Lecture Notes in Computer Science, 176–185. Springer-Verlag.

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-07)*, 305–310. AAAI Press.

Huang, J. 2007. The effect of restarts on the efficiency of clause learning. In Veloso, M., ed., *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 2318–2323. AAAI Press.

Kautz, H., and Selman, B. 1996. Pushing the envelope: planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference*, 1194–1201. AAAI Press.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and graph-based planning. In Dean, T., ed., *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, 318–325. Morgan Kaufmann Publishers.

Larrabee, T. 1992. Test pattern generation using Boolean satisfiability. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 11(1):4–15.

Marques-Silva, J. P., and Sakallah, K. A. 1996. GRASP: A new search algorithm for satisfiability. In *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*, 220–227.

Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th ACM/IEEE Design Automation Conference (DAC'01)*, 530–535. ACM Press.

Pipatsrisawat, K., and Darwiche, A. 2007. A lightweight component caching scheme for satisfiability solvers. In Marques-Silva, J., and Sakallah, K. A., eds., *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2007)*, volume 4501 of *Lecture Notes in Computer Science*, 294–299. Springer-Verlag.

Porco, A.; Machado, A.; and Bonet, B. 2011. Automatic polytime reductions of NP problems into a fragment of STRIPS. In *ICAPS 2011. Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling*, 178–185. AAAI Press.

Rankooh, M. F., and Ghassem-Sani, G. 2013. New encoding methods for SAT-based temporal planning. In *ICAPS 2013. Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, 73–81. AAAI Press.

Rintanen, J., and Gretton, C. O. 2013. Computing upper bounds on lengths of transition sequences. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, 2365–2372. AAAI Press.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2004. Parallel encodings of classical planning as satisfiability. In Alferes, J. J., and Leite, J., eds., *Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004. Proceedings*, number 3229 in Lecture Notes in Computer Science, 307–319. Springer-Verlag.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence* 170(12-13):1031–1080.

Rintanen, J. 2004. Evaluation strategies for planning as satisfiability. In López de Mántaras, R., and Saitta, L., eds., *ECAI 2004. Proceedings of the 16th European Conference on Artificial Intelligence*, 682–687. IOS Press.

Rintanen, J. 2008a. Planning graphs and propositional clause-learning. In Brewka, G., and Doherty, P., eds., *Principles of Knowledge Representation and Reasoning: Proceedings of the Eleventh International Conference (KR 2008)*, 535–543. AAAI Press.

Rintanen, J. 2008b. Regression for classical and nondeterministic planning. In Ghallab, M.; Spyropoulos, C. D.; and Fakotakis, N., eds., *ECAI 2008. Proceedings of the 18th European Conference on Artificial Intelligence*, 568–571. IOS Press.

Rintanen, J. 2009. Planning and SAT. In Biere, A.; Heule, M. J. H.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability*, number 185 in Frontiers in Artificial Intelligence and Applications. IOS Press. 483–504.

Rintanen, J. 2010a. Heuristic planning with SAT: beyond uninformed depth-first search. In Li, J., ed., *AI 2010 : Advances in Artificial Intelligence: 23rd Australasian Joint Conference on Artificial Intelligence, Adelaide, South Australia, December 7-10, 2010, Proceedings*, number 6464 in Lecture Notes in Computer Science, 415–424. Springer-Verlag.

Rintanen, J. 2010b. Heuristics for planning with SAT. In Cohen, D., ed., *Principles and Practice of Constraint Programming - CP 2010, 16th International Conference, CP 2010, St. Andrews, Scotland, September 2010, Proceedings.*, number 6308 in Lecture Notes in Computer Science, 414–428. Springer-Verlag.

Rintanen, J. 2012a. Engineering efficient planners with SAT. In *ECAI 2012. Proceedings of the 20th European Conference on Artificial Intelligence*, 684–689. IOS Press.

Rintanen, J. 2012b. Planning as satisfiability: heuristics. *Artificial Intelligence* 193:45–86.

Robinson, N.; Gretton, C.; Pham, D.-N.; and Sattar, A. 2009. SAT-based parallel planning using a split representation of actions. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *ICAPS 2009. Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, 281–288. AAAI Press.

Sideris, A., and Dimopoulos, Y. 2010. Constraint propagation in propositional planning. In *ICAPS 2010. Proceedings of the Twentieth International Conference on Automated Planning and Scheduling*, 153–160. AAAI Press.

Smith, A.; Veneris, A.; Fahim Ali, M.; and Viglas, A. 2005. Fault diagnosis and logic debugging using Boolean satisfiability. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24(10).

Streeter, M., and Smith, S. F. 2007. Using decision procedures efficiently for optimization. In *ICAPS 2007. Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling*, 312–319. AAAI Press.

Wood, R. G., and Rutenbar, R. A. 1998. FPGA routing and routability estimation via Boolean satisfiability. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 6(2).

Zarpas, E. 2004. Simple yet efficient improvements of SAT based bounded model checking. In Hu, A. J., and Martin, A. K., eds., *Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004. Proceedings*, number 3312 in Lecture Notes in Computer Science, 174–185. Springer-Verlag.

The AllPACA Planner: All Planners Automatic Choice Algorithm

Yuri Malitsky
Insight Center for Data Analytics
University College Cork

David Wang and **Erez Karpas**
Model-based Embedded and Robotic Systems Group
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology

Abstract

The AllPACA planner is a portfolio planner, which automatically chooses which of several planners to run for the planning task that it is given. AllPACA is based on machine learning techniques, which attempt to choose the planner that will result in the fastest solution time, based on the features of the planning task. In the sequential optimal track, AllPACA was pre-trained on all planning tasks the sequential optimal track in all previous editions of the International Planning Competition. For the learning track, AllPACA can also learn to predict planner performance on tasks from each domain during the training phase, and can additionally exploit domain-specific features.

Introduction

The AllPACA[1] planner is a portfolio planner, which automatically chooses which of several planners to run for the planning task that it is given. However, AllPACA differs from most portfolio planners in previous editions of the International Planning Competition. Inspired by the recent successes of algorithm portfolios in SAT, MaxSAT, and CSP Competitions, the new portfolio chooses only one planner to evaluate on a new instance. This decision is made by a prediction model trained based on newly introduced features for the planning task. Ultimately, AllPACA attempts to choose the planner that is expected to solve the given planning task the fastest.

In this paper, we review the AllPACA planner. We begin by discussing the planners that comprise our portfolio, proceeding to discuss the employed features and the machine learning techniques used. We conclude by explaining how AllPACA can be modified to also compete in the learning track of the International Planning Competition.

The Planners

For the sequential optimal track of IPC 2014, we decided to use all 12 planners which participated in the sequential optimal track of IPC 2011 (Coles et al. 2012). As some of these planners share the same codebase, we only had to ship 5 separate planner distributions with AllPACA. Table 1 lists each of the planners that we used, and the paper describing it; all paper are available in the IPC 2011 booklet (García-Olaya, Jiménez, and Linares López 2011).

Planner	Described in
bjolp	Domshlak et al. (2011a)
lmcut	Helmert and Domshlak (2011)
fd-autotune	Fawcett et al. (2011)
fdss-1	Helmert et al. (2011)
fdss-2	Helmert et al. (2011)
merge-and-shrink	Nissim, Hoffmann, and Domshlak (2011)
selmax	Domshlak et al. (2011b)
forkinit	Katz and Domshlak (2011)
iforkinit	Katz and Domshlak (2011)
lmfork	Katz and Domshlak (2011)
cpt4	Vidal (2011)
gamer	Kissmann and Edelkamp (2011)

Table 1: The Planners in AllPACA's Portfolio

Training AllPACA

This section describes the necessary components for choosing which planner to use for each task: the features employed, the set of planning tasks used for training, and the learning algorithm applied.

Features

AllPACA currently relies on mostly syntactic features of the planning task's PDDL description, and of the finite domain representation produced by the Fast Downward translator (Helmert 2009). The PDDL features are divided into features of the domain, and features of the problem. The features of the finite domain representation are extracted from the translator's output on a specific task, and are therefore only features of that task. The complete list of features appears in Figure 1.

Training Set

AllPACA attempts to predict which planner will be the fastest on a given planning task. For training it requires a list of tasks (with features), and the running time of all planners on these tasks. We ran all 12 planners from IPC 2011 on all

[1] AllPACA is closely related to, but not associated with, LAMA.

PDDL

Domain

1 `Number of object types`
2 `Number of predicates`
3 `Number of (lifted) operators`
4-7 `Ratio of add/delete effects`: mean, stdev, min and max
8-11 `Arity of propositions`: mean, stdev, min and max
12-15 `Number of operator preconditions`: mean, stdev, min and max
16-19 `Number of operator add effects`: mean, stdev, min and max
20-23 `Number of operator delete effects`: mean, stdev, min and max
24 `Number of constants`
25 `Number of goal propositions`
26-29 `Arity of goal propositions`: mean, stdev, min and max

Problem

30 `Number of objects`
31 `Number of objects and constants`
32 `Number of initial state facts`
33 `Number of possible grounded propositions`
34-37 `Number of grounded propositions per predicate`: mean, stdev, min and max
38 `Number of total ground actions`
39-42 `Number of ground actions per operator type`: mean, stdev, min and max

FDR

43 `Number of variables`
44 `Number of muteness discovered`
45 `Number of invariants discovered`
46 `Number of goals`
47 `Number of grounded actions`
48 `Number of axions`
49 `Number of grounded actions and axioms`
50-53 `Number of prevail conditions`: mean, stdev, min and max
54-57 `Number of pre post conditions`: mean, stdev, min and max
58-61 `Number of action effects`: mean, stdev, min and max
62-65 `Variable domain sizes`: mean, stdev, min and max

Figure 1: Features employed to describe planning tasks.

planning tasks from the sequential optimal tracks of all previous IPCs, using the IPC 2011 software (López, Celorrio, and Helmert 2013).

Learning Algorithm

A plethora of learning algorithms have been applied to the task of algorithm selection. There are algorithms that learn a regression model over the features to predict the runtime of each solver (Silverthorn and Miikkulainen 2010), choosing the one with the lowest expected time. Others learn forests of trees to distinguish a winner between every pair of solvers, relying on the one that is voted to win most pairings (Xu et al. 2012). Still others cluster instances, assigning a single solver to each group (Malitsky and Sellmann 2012). For a detailed overview of developed techniques we refer the reader to (Kotthoff, Gent, and Miguel 2012).

Surprisingly, despite all of the research into algorithm selection, a recent paper (Hutter et al. 2013) has shown that random forests are among the best predictors of a solver's runtime. Confirming this result internally on the planning task data, AllPACA employs this prediction model for its own decisions. For training, the method is provided with a collection of training instances, feature vectors that describe each instance, and the runtime of each solver on each instance. Random forests, each with 300 trees and a max depth of 3 for each tree, are trained to predict the runtime of each solver. Ultimately, the solver with the best predicted performance is evaluated on the new instance at hand.

The AllPACA Planner

In this section, we tie everything together and describe how AllPACA works at a high level. Figure 2 illustrates the data flow of the AllPACA planner.

In an offline phase (on the left side of Figure 2), we show that AllPACA runs all given planners on a set of training instances, and extracts the features from them. In this submission, we used all planning tasks from all previous IPCS. AllPACA then builds a classifier based upon this information, which predicts which planner will solve a given planning task the fastest.

When given a planning task to solve (top middle of Figure 2), AllPACA first extracts the features from that problem, and uses its classifier to predict which planner will solve it the fastest. It then runs that chosen planner, which hopefully terminates within the given time limit, and provides a solution. However, as we are using third party planners, some of which could provide incorrect solutions, we first check whether the planner produced a valid solution. If it did, we keep that solution. Otherwise, if the planner failed (for example, due to lack of support of conditional effects) or produced an invalid plan, we run a default planner. In this submission, the planner is Fast Downward running the LM-CUT heuristic (Helmert and Domshlak 2009), which has been modified to support conditional effects by further relaxing operators into unary effect operators.

Discussion

In AllPACA, we try to choose the planner which will solve a given planning task the fastest. While this is suitable for the optimal track, as we are only interested in optimal solutions, this is not necessarily the best thing to do for the satisficing track. In a setting where the objective is to find the best possible solution under a given time limit, it is not clear what is the best way to choose a planner. However, we believe that AllPACA decision rule could be used as is, in order to choose the *first* planner to run, as solving a problem is al-

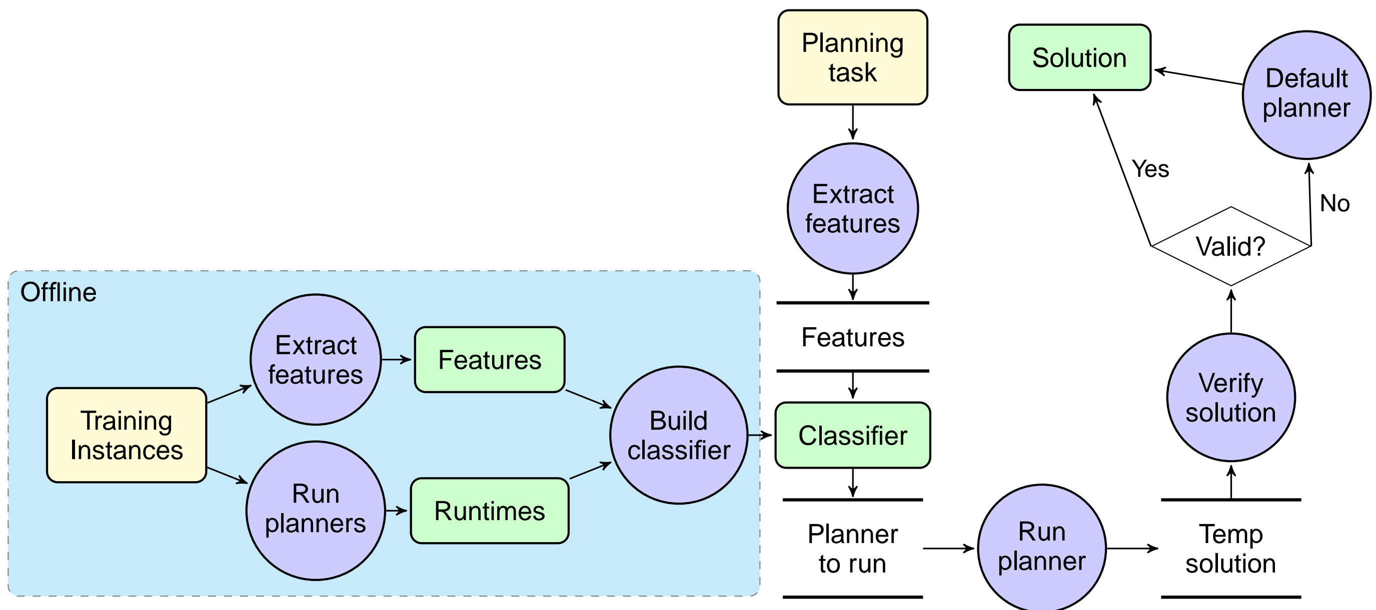


Figure 2: Data Flow Diagram of AllPACA Planner

ways better than not solving it. As a second stage, we would need to choose how to improve the solution that was found, which is the subject of future work.

References

Coles, A. J.; Coles, A.; Olaya, A. G.; Celorrio, S. J.; López, C. L.; Sanner, S.; and Yoon, S. 2012. A survey of the seventh international planning competition. *AI Magazine* 33(1).

Domshlak, C.; Helmert, M.; Karpas, E.; Keyder, E.; Richter, S.; Roger, G.; Seipp, J.; and Westphal, M. 2011a. BJOLP: The big joint optimal landmarks planner.

Domshlak, C.; Helmert, M.; Karpas, E.; and Markovitch, S. 2011b. The SelMax planner: Online learning for speeding up optimal planning.

Fawcett, C.; Helmert, M.; Hoos, H.; Karpas, E.; Roger, G.; and Seipp, J. 2011. Fd-autotune: Automated conguration of fast downward.

García-Olaya, A.; Jiménez, S.; and Linares López, C. 2011. The 2011 international planning competition: Description of participating planners, deterministic track. http://www.plg.inf.uc3m.es/ipc2011-deterministic/attachments/ParticipatingPlanners/ipc2011-booklet.pdf.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. ICAPS 2009*, 162–169.

Helmert, M., and Domshlak, C. 2011. Lm-cut: Optimal planning with the landmark-cut heuristic.

Helmert, M.; Roger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S.; and Westphal, M. 2011. Fast downward stone soup.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *AIJ* 173:503–535.

Hutter, F.; lin Xu; Hoos, H. H.; and Leyton-Brown, K. 2013. Algorithm runtime prediction: Methods and evaluation.

Katz, M., and Domshlak, C. 2011. Planning with implicit abstraction heuristics.

Kissmann, P., and Edelkamp, S. 2011. Improving cost-optimal domain-independent symbolic planning.

Kotthoff, L.; Gent, I.; and Miguel, I. P. 2012. An evaluation of machine learning in algorithm selection for search problems. *AI Communications*.

López, C. L.; Celorrio, S. J.; and Helmert, M. 2013. Automating the evaluation of planning systems. *AI Communications* 26(4):331–354.

Malitsky, Y., and Sellmann, M. 2012. Instance-specific algorithm configuration as a method for non-model-based portfolio generation. *CPAIOR* 244–259.

Nissim, R.; Hoffmann, J.; and Domshlak, C. 2011. The merge-and-shrink planner: Bisimulation-based abstraction for optimal planning.

Silverthorn, B., and Miikkulainen, R. 2010. Latent class models for algorithm portfolio methods. *AAAI*.

Vidal, V. 2011. Cpt4: An optimal temporal planner lost in a planning competition without optimal temporal track.

Xu, L.; Hutter, F.; Shen, J.; Hoos, H. H.; and Leyton-Brown, K. 2012. Satzilla2012: Improved algorithm selection based on cost-sensitive classification models. SAT Competition.

cGamer: Constrained Gamer

Álvaro Torralba and **Vidal Alcázar**
{alvaro.torralba, vidal.alcazar}@uc3m.es
Universidad Carlos III de Madrid, Madrid, Spain

Peter Kissmann
kissmann@cs.uni-saarland.de
Saarland University, Saarbrücken, Germany

Stefan Edelkamp
edelkamp@tzi.de
University of Bremen, Bremen, Germany

Abstract

Gamer is a symbolic planner that performs (in this case) bidirectional symbolic search. It already participated in 2008 and 2011, so in this paper we will focus on the improvements since then. The main improvements are: fixing some bugs and implementing basic improvements; a disjunctive partitioning of the transition relations; and the exploitation of state invariants both during the preprocessing phase and during search.

Motivation

Symbolic search can obtain exponential savings in both time and space compared to regular explicit-state search (McMillan 1993). This is achieved by using binary decision diagrams (BDDs) (Bryant 1986) to represent both sets of states and transition relations (TRs). Furthermore, the use of BDDs allows a seamless implementation of detection of subsumed states and collision of frontiers, which opens up the possibility of using regression and bidirectional search algorithms efficiently (Alcázar, Fernández, and Borrajo 2014).

Gamer (Kissmann and Edelkamp 2011), the vanilla version of cGamer, already participated in the International Planning Competitions (IPC) of 2008 and 2011. While it won the competition in 2008, in 2011 it solved 148 problems, which was kind of underwhelming given that the winner solved 185 tasks. As described in Section 6.5.5 of Peter Kissmann's PhD Thesis (Kissmann 2012), after fixing some bugs and implementing some basic improvements (such as using a more efficient parser for grounded PDDL) Gamer is able to solve 13 additional problems (own results). Also, on a *per domain* analysis we can see that Gamer, although still worse overall, outperforms the rest of the participants in quite a few domains. All this means that symbolic search cannot be ruled out and that Gamer, under the right circumstances, is still a strong contestant.

Driven by recent research, an important increase in performance in Gamer has been obtained. Two orthogonal works, one dealing with the handling of the TRs (Torralba, Edelkamp, and Kissmann 2013) and another dealing with the encoding of state invariants in symbolic search (Torralba and Alcázar 2013), reported significantly better coverage than the one obtained by Gamer. cGamer, the planner described here, implements some techniques proposed in these previous works.

Symbolic Search

Symbolic search was originally proposed in the area of model checking (McMillan 1993). The basic idea is to perform set-based search as opposed to the traditional search expanding one state at a time. Sets of states are represented with efficient data structures, like Binary Decision Diagrams (BDDs) (Bryant 1986). To perform the search, planning actions are represented with one or more Transition Relations (TRs). The successor generation in symbolic search is performed with the *image* and *pre-image* operations of a set of states with respect a transition relation. Given a set of states and a TR, the *image* operation computes the set of successor states that can be reached from any state in the set by applying any operator represented by the TR. Similarly, the *pre-image* operation computes the set of predecessor states in regression.

Our predecessor planner, Gamer, is a symbolic search planner that implements two algorithms: symbolic breadth-first search and symbolic A^* with symbolic pattern database heuristics (Culberson and Schaeffer 1998; Edelkamp 2002). From what we observed both in the results of IPC-11 and our experimentation, using pattern databases (PDBs) is often worse than just using bidirectional blind search. This is the case for most domains without non-unit action costs, and even in unit-cost domains selecting patterns and precomputing the PDBs is often not better than just searching in both directions with no heuristics.

The main advantage of the bidirectional search is that it can interleave the search in forward and backward directions. Gamer estimates which direction is easier to expand by taking into account the time spent in the last steps. Thus, the new version of Gamer features a symbolic bidirectional Dijkstra search, that is able to cope with non-unit cost domains (Edelkamp, Kissmann, and Torralba 2012).

Preprocessing

Apart from using Gamer's parser, we employ Fast Downward's translator and preprocessor (Helmert 2006; 2009) to generate the SAS^+ encoding of the task.[1] There are

[1] Using both Gamer and Fast Downward in the preprocessing phase is redundant. We did so for convenience, as not all the necessary techniques were implemented in both preprocessors.

three noteworthy considerations regarding the preprocessing phase in cGamer:

- How the SAS$^+$ variables are selected.
- How to compute h^2 (Bonet and Geffner 2001) and prune spurious operators.
- How conditional effects are dealt with.

SAS$^+$ Variable Selection

Switching from Gamer's SAS$^+$ encoding to the Fast Downward version (Helmert 2009), we observed a decrease of performance in some benchmark domains. We changed the selection of which invariant groups are used as SAS$^+$ variables in order to avoid that degradation in performance.

The Fast Downward planner chooses invariant groups with the highest cardinality as SAS$^+$ variables, until all the fluents of the problem have been considered in a variable. Aiming to further reduce the number of SAS$^+$ variables selected, we prefer to select invariant groups that contain fluents that do not appear in other invariant groups. We base our criterion on the observation that, since all the fluents of the problem have to be included in a SAS$^+$ variable, invariant groups that have a fluent which does not appear in other invariant groups will always be selected anyway.

As an example, think of the following case, based on a simplified version of the IPC-2011 *floortile* domain: we have two robots on a grid such that the robots cannot be at the same cell at the same time. Two types of invariant groups are detected:

1. Each robot is at exactly one single cell:
   (at robot1 cell1),(at robot1 cell2),...
2. Each cell either is clear or has a robot at it:
   (clear cell1),(at robot1 cell1),(at robot2 cell1)

Invariants of the first type have larger cardinality, so Fast Downward would encode this problem with a variable per robot that represents the location of the robot ({*(at robot1 cell1),(at robot1 cell2),...* }) and a variable of the kind {*(clear cell1),⟨none of those⟩*} per cell. In our case, we prefer to select invariant groups of the second type first because each fluent *(clear cell1)* only takes part on a single invariant group. Thus, we would only have a variable per cell of the kind {(clear cell1),(at robot1 cell1),(at robot2 cell1)}, which amounts to fewer variables and fluents.

This leads to the use of *"exactly-one"* invariant groups as variables in most cases, avoiding the use of *"at-most-one"* invariant groups if possible – which require an additional ⟨*none of those*⟩ fluent. With this policy the number of resulting variables and fluents is usually lower. This may be counterproductive if techniques that depend on the causal graph are used, but this only affects us when choosing the ordering of the variables.

Computing h^2 Invariants and Pruning Spurious Operators

We have implemented the computation of the h^2 in Fast Downward's preprocessor. We also implemented a backward version of h^2 (Haslum 2008), which identifies pairs of propositions that cannot be reached from goal states in regression.

We use the mutexes obtained from h^2 and the *"exactly-one"* invariant groups from Fast Downward's monotonicity analysis to disambiguate the preconditions and the effects of the operators of the problem (Alcázar et al. 2013). We discard operators whose preconditions or effects are spurious sets of fluents, that is, contradict the previously inferred state invariants. We do this because the number of ground operators is significantly reduced in many planning domains with respect to the standard preprocessor of Fast Downward.

The discovery and use of the state invariants during this phase is interleaved: whenever new mutexes or spurious operators are discovered in this process, we repeat the computation of h^2 in both directions and the operator disambiguation until no more constraints are inferred. We set a limit of 300 seconds for this phase.

Conditional Effects

Conditional effects are compiled away by using *adl2strips* (Hoffmann et al. 2006), a tool initially developed by Jörg Hoffmann and modified later by Sergio Núñez that converts more expressive planning instances into STRIPS. *adl2strips* implements two different methods for compiling away conditional effects (Gazen and Knoblock 1997; Nebel 2011). Both compilations have their pros and cons. Gazen and Knoblock's compilation may generate an exponential number of STRIPS actions on the size of the input task. On the other hand, Nebel's compilation guarantees that the number of STRIPS actions is polynomial on the size of the input task, but increases the plan length. Therefore, we use Gazen and Knoblock's compilation for tasks with at most three conditional effects in the same PDDL action and Nebel's compilation otherwise.

Disjunctive Transition Relations

Torralba, Edelkamp, and Kissmann (2013) identified the *image* and *pre-image* operations and the subsequent union of successor sets as a bottleneck. In other words, the encoding of planning operators in the Transition Relations (TRs) may have a large impact on the overall performance. In Gamer, a TR was used to represent each operator. Several alternatives were proposed; among them, computing a disjunction of the TRs of operators with the same cost stood out for its simplicity and results.

Since the disjunction of all the TRs of the problem is sometimes not tractable to compute, we set a limit MAX_TR_SIZE on the maximum number of nodes that any TR may have. If this limit is reached during the computation of the disjunction of TRs, several disjunctive TRs smaller than MAX_TR_SIZE will be used instead of a single TR. When multiple disjunctive TRs must be used, the choice of which TRs must be merged is critical, as the number and size of the resulting disjunctive TRs depends on it. This is done using a balanced merging approach based on the disjunction tree used in the original Gamer to compute the disjunction of successor sets. For the competition, cGamer uses a limit of MAX_TR_SIZE=100k nodes.

Encoding State Invariants in Symbolic Search

Regression is common in symbolic search, both as a part of the main search algorithm and as a way to derive admissible heuristics. Although constraints derived from state invariants are commonly used in explicit-state regression, in symbolic regression this had not been done. Hence, Torralba and Alcázar (2013) proposed several ways of encoding these constraints in symbolic search. According with the experimental results, the most efficient method to apply constraints in symbolic search is to encode them in the TRs.

In cGamer we encode binary mutexes derived from the computation of h^2 and invariant groups derived from Fast Downward's monotonicity analysis. Before computing the disjunction of the TRs described in the previous section, the TR corresponding to each operator is enriched with all the constraints that may be violated after applying the operator. Thus, no state violating the constraints is generated in the search.

An important difference with respect to the version presented in (Torralba and Alcázar 2013) is that we also derive mutexes from the backward computation of h^2. These backward mutexes are used to prune the forward search in a similar manner to how forward h^2 mutexes are used in regression.

Acknowledgements

This work has been partially supported by a FPI grant from the Spanish government associated to the MICINN project TIN2008-06701-C03-03, and it has also been supported by the project TIN2011-27652-C03-02. We'd like to thank both Jörg Hoffmann and Sergio Núñez for *adl2strips*.

References

Alcázar, V.; Borrajo, D.; Fernández, S.; and Fuentetaja, R. 2013. Revisiting regression in planning. In *International Joint Conference on Artificial Intelligence*, 2254–2260.

Alcázar, V.; Fernández, S.; and Borrajo, D. 2014. Analyzing the impact of partial states on duplicate detection and collision of frontiers. In *International Conference on Automated Planning and Scheduling*.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

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. *Comput. Intell.* 14(3):318–334.

Edelkamp, S.; Kissmann, P.; and Torralba, Á. 2012. Symbolic A* search with pattern databases and the merge-and-shrink abstraction. In *European Conference on Artificial Intelligence (ECAI)*, 306–311.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *Conference on Artificial Intelligence Planning Systems (AIPS)*, 274–283.

Gazen, B. C., and Knoblock, C. A. 1997. Combining the expressivity of ucpop with the efficiency of graphplan. In *ECP*, 221–233.

Haslum, P. 2008. Additive and reversed relaxed reachability heuristics revisited. *Proceedings of the 6th International Planning Competition*.

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

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5-6):503–535.

Hoffmann, J.; Edelkamp, S.; Thíebaux, S.; Englert, R.; Liporace, F.; and Trüg, S. 2006. Engineering benchmarks for planning: the domains used in the deterministic part of IPC-4. *Journal of Artificial Intelligence Research (JAIR)* 26:453–541.

Kissmann, P., and Edelkamp, S. 2011. Improving cost-optimal domain-independent symbolic planning. In *AAAI Conference on Artificial Intelligence (AAAI)*, 992–997.

Kissmann, P. 2012. *Symbolic Search in Planning and General Game Playing*. Ph.D. Dissertation, Universität Bremen, Germany.

McMillan, K. L. 1993. *Symbolic Model Checking*.

Nebel, B. 2011. On the compilability and expressive power of propositional planning formalisms. *CoRR* abs/1106.0247.

Torralba, Á., and Alcázar, V. 2013. Constrained symbolic search: On mutexes, BDD minimization and more. In *Symposium on Combinatorial Search (SoCS)*, 175–183.

Torralba, Á.; Edelkamp, S.; and Kissmann, P. 2013. Transition trees for cost-optimal symbolic planning. In *International Conference on Automated Planning and Scheduling*.

Gamer and Dynamic-Gamer – Symbolic Search at IPC 2014

Peter Kissmann
Saarland University
Saarbrücken, Germany
kissmann@cs.uni-saarland.de

Stefan Edelkamp
Universität Bremen
Bremen, Germany
edelkamp@tzi.de

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

Abstract

After its success in IPC 2008, Gamer fared much worse in IPC 2011. Nevertheless, it is still the state-of-the-art symbolic search planner, and even in IPC 2011 it was obvious that this kind of search has its merits as in some domains Gamer was able to find more solutions than all other planners.

We participate in IPC 2014 with two different versions of this planner. One version, called Gamer, is a straight-forward extension of the IPC 2011 version, fixing some bugs, improving some code fragments, supporting conditional effects, and replacing the complicated handling of (partial) pattern databases by a bidirectional symbolic version of Dijkstra's algorithm. The other version, called Dynamic-Gamer, extends this code further by making use of dynamic variable reordering.

Introduction

In recent years, the most successful approaches for automated planning have made use of explicit A* search using heuristic functions to evaluate states (see, e. g., (Haslum and Geffner 2000; Helmert et al. 2014; Helmert and Domshlak 2009; Helmert et al. 2011)). The idea here is to have heuristics to guide the search in order to generate fewer states. Generating fewer states means that (a) the runtime decreases (at least if the overhead of the heuristic calculations is small), and (b) the memory requirements decrease.

An alternative approach, though admittedly currently only rarely used, is to apply symbolic search (McMillan 1993). This does not expand all states separately as is done in explicit search, but rather expands entire sets of states simultaneously. These sets are represented by means of binary decision diagrams (BDDs) (Bryant 1986), which are much more memory efficient than explicitly representing each state of the given set.

Our symbolic search planner Gamer (Edelkamp and Kissmann 2009; Kissmann and Edelkamp 2011) started out in the international planning competition (IPC) 2008 and was able to win the competition. However, in the instance of 2011 it ended up close to the end of the field of participants. Nevertheless, from the results it is obvious that symbolic search still has its merits, as in several domains Gamer was able to find optimal solutions that no other planner could find.

For this year's instance of the IPC we extended Gamer in several ways. The first are merely a number of improvements on top of the previous version, overcoming problems we identified in the aftermath of IPC 2011. In addition, we also implemented some real extensions. The most notable ones are the use of bidirectional Dijkstra search instead of symbolic A* search with symbolic pattern databases, the handling of conditional effects and, in case of Dynamic-Gamer, use of dynamic reordering of the BDD variables at runtime for a limited time.

In the following we will start with providing some of the necessary background in planning and symbolic search. Then we will briefly discuss the code improvements before we turn to the actual extensions. Finally, we will provide some results comparing the three versions of Gamer: the one of IPC 2011, and Gamer and Dynamic-Gamer as submitted to IPC 2014.

Background

Planning

We consider planning tasks in finite-domain representation (FDR). Such a task is a tuple $\Pi = \langle V, A, I, G, c \rangle$, where V is a set of finite-domain state variables, where each $v \in V$ has an associated domain $D(v)$. A is a finite set of actions, where each $a \in A$ is a pair $\langle pre_a, \mathit{eff}_a \rangle$, both being partial assignments to V; pre_a is called the precondition and eff_a the effect of action a. I is the initial state and is given by a complete assignment to V. G is a description of the goal states in form of a partial assignment to V. Finally, $c : A \mapsto \mathbb{N}^+$ is a function assigning a cost to each action. We denote those variables $v \in V$ of a partial assignment pa for which $pa(v)$ is defined by $\mathcal{V}(pa)$.

We say that an action $a \in A$ is applicable in state s if $pre_a \subseteq s$. The successor s' reached by applying action a in state s is specified by

$$s'(v) = \begin{cases} \mathit{eff}_a(v) & \text{for all } v \in \mathcal{V}(\mathit{eff}_a) \\ s(v) & \text{for all } v \in V \setminus \mathcal{V}(\mathit{eff}_a). \end{cases}$$

A solution (here called a plan) of a planning task is a sequence of actions $P = (a_1, \dots, a_n)$ that can be applied in the initial state and leads to a goal state s_n: each action a_i must be applicable in state s_{i-1} and results in the successor state s_i with $s_0 = I$ and $G \subseteq s_n$. The cost of a

plan is the sum of the costs of all actions in the plan, i. e., $c(P) = \sum_{i=1}^{n} c(a_i)$. A plan is called optimal if no other plan has lower cost. In this paper we are only concerned with optimal planning.

Symbolic Search

The term *symbolic search* originates in the model checking literature (McMillan 1993). The basic idea is to perform set-based search as opposed to the traditional search expanding one state at a time. The datastructure used to represent such sets of states are binary decision diagrams (BDDs) (Bryant 1986).

A binary decision diagram is a directed acyclic graph, where each internal node has two outgoing edges (the 0 and the 1 edge) and represents the assignment to a corresponding variable. The two terminal nodes are called sinks, and denoted with a Boolean value (0 or 1). A BDD basically represents the satisfying assignments of the represented Boolean formula. However, given that we can represent states by binary variables[1], for search algorithms any such satisfying assignment can be seen as a state represented by the BDD.

Any path leading from the root to the 1 sink and the resulting assignment to the visited variables corresponds to a satisfying assignment (and thus to a state represented by the BDD). The unvisited variables can take any value. Paths leading to the 0 sink correspond to unsatisfied assignments (and thus to states not represented by the BDD).

What we actually mean when talking about BDDs are reduced ordered BDDs (ROBDDs). These have a fixed variable ordering on all paths from the root to the sinks. This allows to apply two reduction rules: (a) remove any node having the same successor along both outgoing edges; (b) merge any two nodes representing the same variable that have the same successor along the 0 edge and the same successor along the 1 edge. Applying these rules results in a canonical representation.

In order to perform symbolic search, we need two sets of variables. The first one, x, represents the current state variables, the second one, x', represents the successor state variables. With these we can represent a transition relation as follows. For each action $a \in A$, we can create a BDD

$$T_a(x, x') = pre_a(x) \wedge \mathit{eff}_a(x') \wedge biimp(V \setminus \mathcal{V}(\mathit{eff}_a), x, x')$$

with $biimp(V', x, x') = \bigwedge_{v \in V'} v(x) \leftrightarrow v(x')$. The full (monolithic) transition relation T is then $T(x, x') = \bigvee_{a \in A} T_a(x, x')$. However, representing this typically takes too much memory (and also too much runtime for the operators making use of it), so that we store the BDDs for the actions' transition relations separately (Burch, Clarke, and Long 1991).

With such a transition relation, given a set of states S (represented as a BDD), we can then calculate the successor set S' by means of the *image* operator:

$$S'(x) = img(S, T) = \exists x.(S(x) \wedge T(x, x'))[x \leftrightarrow x']$$

[1]A finite-domain variable v with domain $D(v)$ can be represented by a binary counter, consisting of $\log |D(v)|$ binary variables.

The conjunction makes sure that only applicable actions are considered, and that the corresponding successor variables are set. Quantifying the current state variables and swapping the sets of variables (denoted by $[x \leftrightarrow x']$) ensures that the successor states are again represented in the current state variables. With the separate transition relations, the image is equivalently defined as

$$S'(x) = img(S, T) = \bigvee_{a \in A} \exists x.(S(x) \wedge T_a(x, x'))[x \leftrightarrow x'].$$

Instead of explicitly applying the conjunction followed by existential quantification, most BDD packages contain a more efficient implementation of this so called *relational product*, where the quantification and conjunction are intertwined (Burch et al. 1994).

For a set of states S, the set of predecessor states S^- can be calculated similarly, by means of the *preimage* operator:

$$S^-(x') = preimg(S, T) = \bigvee_{a \in A} \exists x'.(S(x' \wedge T_a(x, x'))[x \leftrightarrow x']$$

Concerning the variable ordering, recall that we are concerned with finite domain representations. However, these can be easily encoded by binary variables: For a finite domain variable with domain $D(v)$ we need $\log |D(v)|$ binary variables (similar to binary counters) and can map each of the values of the FD variable to one assignment of these binary variables. In the ordering, the variables representing the same FD variable are always kept together. Additionally, the pairs of x and x' variables are stored in an interleaved fashion (Burch et al. 1994).

Symbolic Blind Search Using the image operator, breadth-first search is straight-forward. All we need to do is start with the BDD representing the initial state and then apply the image operator repeatedly to the result until a goal state is reached, i. e., until a BDD S is reached for which $S(x) \wedge G(x) \neq 0$.

Thanks to the separate transition relations, we can also easily cope with action costs. For this we can define an image operator img_c, which takes only those transitions into account that are based on actions having the specified cost c. Thus, symbolic Dijkstra search is performed as follows: We store all states having the same distance from the initial state (g) in a BDD. For a given set of states with g value g_s, all successors based on actions with cost c are generated by means of the img_c operator and added to the BDD representing the states with g value $g_s + c$ by applying disjunction. As soon as a BDD containing goal states is to be expanded, we can stop the search and retrieve a solution, which corresponds to an optimal plan.

Symbolic A* Search Two versions of symbolic A* search have been proposed: BDDA* by Edelkamp and Reffel (1998) and SetA* by Jensen, Bryant, and Veloso (2002). Our implementation follows the basic notions of these approaches.

We construct a two-dimensional matrix of BDDs, where one dimension corresponds to the g values, and the other to a heuristic estimate of the distance to a goal state, i. e., the

h values. We expand those buckets with smallest f value, and among those the one with smallest g value, first. The successors are inserted according to their new g values and the heuristic estimate. If we use a consistent heuristic, this order guarantees that each BDD is expanded at most once, i. e., it is never necessary to expand (g, h) buckets that were already expanded in a previous step. A consistent heuristic means that the f values of the successors can never be smaller than the one of the currently expanded states. Additionally, with each expansion the g value can only increase. Thus, if the successor states have the same f value they must be further along the current f diagonal and will be expanded later and not inserted into an already expanded bucket.

Gamer

In 2008, Gamer (Edelkamp and Kissmann 2009) automatically decided between two different algorithms to run: In case of non-uniform action costs it performed symbolic A* search, while in case of only uniform costs it used symbolic bidirectional breadth-first search. As the heuristic we decided to use pattern databases (PDBs) (Culberson and Schaeffer 1998), whose extension to the symbolic search setting is due to Edelkamp (2002). As we did not have a good algorithm for automatically deciding on a pattern we actually used partial PDBs (Anderson, Holte, and Schaeffer 2007), and used the full variable set as the pattern – in other words, we basically performed bidirectional Dijkstra search, in a non-interleaved fashion, where we allowed half the available time, i. e., half of 30 minutes, for the backward search. The remaining time was then spent in forward direction running symbolic A*. For breadth-first search we implemented an interleaved approach, which automatically decided for each step whether to continue in forward or backward direction, based on the times needed for the previous steps in both directions.

For IPC 2011, we held on to this basic scheme, with three extensions (Kissmann and Edelkamp 2011):

1. automatic calculation of patterns and the corresponding PDBs
2. automatic decision whether to use the automated pattern calculation or the full variable set (in case of non-uniform costs); and whether to use uni- or bidirectional breadth-first search (in case of uniform costs)
3. variable ordering based on the causal graph

Again, we spent up to 15 minutes for the PDB generation, and then took the PDB with highest average heuristic value to be used in forward search.

The idea of the ordering of the variables was that variables that depend on each other, as identified by the causal graph, should be ordered close together in order to decrease BDD sizes and thus improve performance.

Extensions to Gamer

Basic Improvements

In the aftermath of IPC 2011 we noticed several shortcomings, detailed in (Edelkamp, Kissmann, and Torralba 2012). Apart from some bug fixes, we decided to fully replace the parser for grounded PDDL input, as in some cases parsing took longer than the available time. Also, we noticed that the way we decided between uni- and bidirectional search could result in a bottleneck: In some domains the first backward step, on which we based the final decision, took longer than the available time, so that we never started the actual search. This we fixed by setting a timeout for this first backward step and stopping the BDD calculations for it in case of timeout (and using only forward search afterwards).

Bidirectional Dijkstra Search

Apart from these small changes we also checked the used algorithms again. It turned out that (a) in many cases the full pattern was chosen, and (b) in cases that an actual abstraction was performed, a large number of PDBs had to be generated, which typically took the allowed time of 15 minutes, leaving only 15 minutes for forward search. The advantage of the bidirectional breadth-first search we were using in the uniform-cost domains was that here the decision between forward and backward search was made dynamically at runtime and the steps could be performed in an interleaved fashion. In domains where forward and backward search result in roughly the same performance the approach we chose for symbolic A* with the fixed time for backward search should result in good performance. However, in domains where one direction tends to be much more difficult than the other an interleaved search would be highly preferable.

Thus, we decided to implement a bidirectional version of Dijkstra search (Edelkamp, Kissmann, and Torralba 2012) (cf. Algorithm 1). Similar to bidirectional breadth-first search this automatically decides in which direction to perform the next step, solely based on the runtimes for the previous steps. For bidirectional Dijkstra search, it might not be immediately obvious how to decide when the search is actually finished, i. e., when a found solution is optimal. However, this question was previously tackled in external bidirectional Dijkstra search (Goldberg and Werneck 2005). The idea is to store the cost c_{total} of the cheapest plan found so far, and stop the search as soon as the sum of the g values of the states to be expanded next in the two directions is at least as high as c_{total}. The same criterion works in symbolic search.

Dynamic Reordering

In a recent study (Kissmann and Hoffmann 2013) we analyzed how closely related the definition of dependence as found by causal graphs and that required to reduce BDD sizes actually are. The result is that there is not much of a relation at all. In fact, even for very limited causal graphs we found that there are families of planning tasks where in some layer the ordering we imposed can result in BDDs containing exponentially more nodes than a minimal one.

Apart from this theoretical result we also evaluated different ordering schemes experimentally against purely random orders. Here the result is that most "informed" orders result in higher coverage, but none fully dominates the random orderings in all domains.

As a delimiter to decide how much better we can get we compared the orderings against the result of running dy-

Algorithm 1 Bidirectional symbolic Dijkstra search, from (Edelkamp, Kissmann, and Torralba 2012)

Input: $\Pi = \langle V, A, I, G, c \rangle$: planning task.
T: set of transition relations for all actions.
Output: cost-optimal plan P.
$P \leftarrow$ "no plan"
$fClosed \leftarrow bClosed \leftarrow 0$
$fReach_0 \leftarrow I$
$bReach_0 \leftarrow G$
$g_f \leftarrow g_b \leftarrow 0$
$c_{total} \leftarrow \infty$
while $g_f + g_b < c_{total}$
  if $NextDirection = Forward$
    $\{g_1, \ldots, g_n\} \leftarrow fStep(fReach, g_f, c, fClosed, bClosed)$
    for all $g \in \{g_1, \ldots, g_n\}$
      for all $\{i \mid i < g_b \wedge bReach_i \neq 0 \wedge g + i < c_{total}\}$
        if $fReach_g \wedge bReach_i \neq 0$
          $c_{total} \leftarrow g + i$
          update P
    $g_f \leftarrow g_f + 1$
  else // same in backward direction
return P

Procedure $fStep(fReach, g, fClosed, bClosed)$
  $Ret \leftarrow \{\}$
  $fReach_g \leftarrow fReach_g \wedge \neg fClosed$
  for all $c \in \{1, \ldots, \max_{a \in A} c(a)\}$
    $Succ \leftarrow \bigvee_{a \in A, c(a) = c} img(fReach_g, T_a)$
    if $Succ \wedge bClosed \neq 0$
      $Ret \leftarrow Ret \cup \{g + c\}$
    $fReach_{g+c} \leftarrow fReach_{g+c} \vee Succ$
  $fClosed \leftarrow fClosed \vee fReach_g$
  return Ret

namic reordering for the whole search, for which we assumed that the BDD sizes should be much closer to optimal.

For the dynamic reordering we made use of Rudell's (1993) sifting algorithm. This starts with the BDD variable with the greatest number of nodes in the current BDD, and then moves it first to the end then to the beginning of the ordering. This is done by swapping the position with the next variable in the corresponding direction. Afterwards it is moved back to the position where the BDD size was smallest. This is repeated for all variables.

Dynamic reordering is automatically executed whenever a threshold in BDD sizes is passed. This threshold is initially set to 4,000 nodes, and after each reordering updated to twice the number of nodes still in the BDDs.

The result of this experiment was pretty clear (cf. Figure 1): In very few tasks the static ordering schemes we tried result in BDDs of slightly smaller size than achievable by dynamic reordering, while in the vast majority of tasks the sizes are much greater, in some cases up to three orders of magnitude. However, this advantage in size comes at a cost, namely runtime. In order to get a sufficient number of dat-

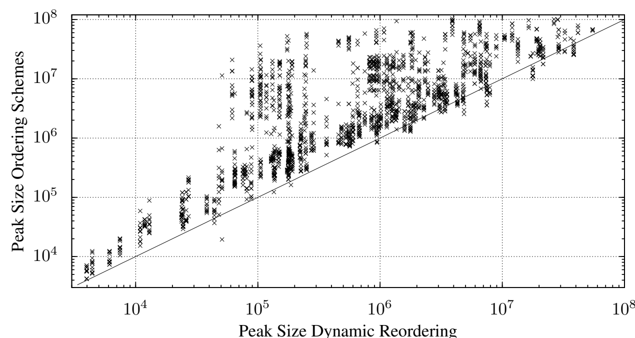


Figure 1: Comparison of BDD sizes using dynamic reordering (x-axis) and static ordering schemes (y-axis). Taken from (Kissmann and Hoffmann 2013).

apoints we had to run the planner using dynamic reordering much longer than the other approaches. When restricting the search time to the usual 30 minutes the coverage was much worse than with the static schemes.

What we learned from this is that using dynamic reordering can be great in order to reduce BDD sizes, but it should not be run for the entire time. In a subsequent study we tried two different approaches: (a) starting dynamic reordering immediately and (b) starting it only after the basic BDDs for the initial and goal states as well as the transition relations have been built. We allowed only fixed numbers of reorderings and tried to find criteria to automatically decide when to stop reordering, so that the overhead in runtime was not too high while the memory savings (and thus an increase in coverage) were visible.

It turned out that the much more stable approach was to start dynamic reordering only after the basic BDDs have been built. As criteria for stopping reordering and sticking to the last generated order we noticed that the best point often was when (a) the quotient between the time for the last reordering and the previous one reaches a certain threshold (often between 1.25 and 1.75), or (b) the percentage of the time spent in the last reordering step on the total runtime increased compared to the previous one.

Grounding

For grounding the planning tasks, from the start Gamer made use of the grounding utility of the MIPS planner (Edelkamp and Helmert 2001). This not only grounds the task at hand but also transforms the task to a finite-domain representation (Edelkamp and Helmert 1999). To do so, it finds mutually exclusive predicates and groups them to construct the FD variables. However, for this check it considers only predicates that agree in one argument. For example, in some logistics task, predicates $at(t_1, c_1)$ and $at(t_1, c_2)$ might be mutually exclusive, denoting the different locations of truck t_1, and can thus be grouped to one FD variable.

We extended this to also check for predicates that agree in two arguments. For example, in some task where different objects might be placed on the cells of a board, $on(s_1, x_1, y_1)$ and $on(s_2, x_1, y_1)$ might be mutually exclusive, denoting the different things that can be placed on the

cell at position (x_1, y_1), and can thus be grouped to one FD variable. This can result in a more concise state representation, as it allows for more facts to be grouped together to FD variables and thus might reduce the encoding size even further.

Additionally we noted that in the old version of the grounding utility only one encoding was generated, without any regard of the size of it. We extended this in the following way. If the number of possible encodings is small, we generate all of them; if the number is too large, we generate as many as possible (in random order) in 1 second. In both cases we return the one with smallest binary encoding size.

Conditional Effects

IPC 2014 is the first instance of the international planning competition that requires all planners to support conditional effects. A conditional effect basically is a kind of sub-action within an action, having its own additional precondition and effect. Thus, an action $a \in A$ with conditional effects consists of the precondition pre_a and effect eff_a as before, but additionally has a set of pairs consisting of conditions and effects: $cond_a = \{\langle cpre_a^1, ceff_a^1\rangle, \ldots, \langle cpre_a^n, ceff_a^n\rangle\}$, where each $cpre_a^i$ and $ceff_a^i$ is a partial assignment to V. An action a with conditional effects is evaluated as follows: If $pre_a \subseteq s$, then the action is applicable in state s. Applying a in s results in a new state s' with

$$s'(v) = \begin{cases} eff_a(v) & \text{if } v \in \mathcal{V}(eff_a) \\ e(v) & \text{if } \langle p, e\rangle \in cond_a, p \subseteq s, v \in \mathcal{V}(e) \\ s(v) & \text{else.} \end{cases}$$

One way to handle conditional effects is to compile them away (see, e. g., (Gazen and Knoblock 1997; Hoffmann et al. 2006; Nebel 2000)). To do so, each action a containing some conditional effect $\langle cpre_a^i, ceff_a^i\rangle$ can be replaced by two actions a'_+ and a'_-. Action a'_+ denotes the resulting action where the condition $cpre_a^i$ is satisfied:

$$\begin{aligned} pre_{a'_+} &= pre_a \cup cpre_a^i \\ eff_{a'_+} &= eff_a \cup ceff_a^i \\ cond_{a'_+} &= cond_a \setminus \{\langle cpre_a^i, ceff_a^i\rangle\} \end{aligned}$$

Action a'_-, on the other hand, denotes the resulting action where $cpre_a^i$ is not satisfied:

$$\begin{aligned} pre_{a'_-} &= pre_a \cup \overline{cpre_a^i} \\ eff_{a'_-} &= eff_a \\ cond_{a'_-} &= cond_a \setminus \{\langle cpre_a^i, ceff_a^i\rangle\} \end{aligned}$$

where $\overline{cpre_a^i}$ denotes the negation of $cpre_a^i$. This replacement can be repeated on the resulting actions until no conditional effects remain. However, due to this each action is replaced by a number of actions exponential in the number of conditional effects the original action has.

As our grounding utility already supports conditional effects, we decided to integrate support into the planner itself and thus prevent this blowup. In order to do so we had to extend the creation of the transition relations. We start with a BDD capturing the precondition and basic effects of an action a:

$$T_a^1(x, x') = pre_a(x) \wedge eff_a(x')$$

Let $\mathcal{V}(cond_a) = \{v \mid \langle p, e\rangle \in cond_a, v \in \mathcal{V}(e)\}$ be the set of all variables that appear in effects of conditional effects of action a. For each variable $v \in \mathcal{V}(cond_a)$ we first determine the set of conditional effects $cond_a(v) \subseteq cond_a$ which have this variable in the effect: $cond_a(v) = \{\langle p, e\rangle \mid \langle p, e\rangle \in cond_a, v \in \mathcal{V}(e)\}$. Then we create the corresponding BDD

$$T_a^2(v, x, x') = \left(\bigvee_{(p,e) \in cond_a(v)} (p(x) \wedge e(v)(x'))\right) \vee \left(\left(\bigwedge_{(p,e) \in cond_a(v)} \neg p(x)\right) \wedge (v(x) \leftrightarrow v(x'))\right).$$

For each variable $v \in V \setminus (\mathcal{V}(eff_a) \cup \mathcal{V}(cond_a))$, we create a BDD

$$T_a^3(v, x, x') = v(x) \leftrightarrow v(x')$$

to model the frame. Finally, the full transition relation for action a is

$$\begin{aligned} T_a(x, x') = & T_a^1(x, x') \wedge \\ & \bigwedge_{v \in \mathcal{V}(cond_a)} T_a^2(v, x, x') \wedge \\ & \bigwedge_{v \in V \setminus (\mathcal{V}(eff_a) \cup \mathcal{V}(cond_a))} T_a^3(v, x, x'). \end{aligned}$$

Evaluation of Different Versions of Gamer

We have submitted two versions of Gamer for IPC 2014, one called *Gamer* as before, the other called *Dynamic-Gamer*. Gamer performs bidirectional symbolic Dijkstra search for all tasks with the extensions as introduced in the previous sections, while Dynamic-Gamer additionally makes use of dynamic reordering. The criteria to stop are when the quotient of the last reordering time and the previous one is at least 1.5, or when the percentage criterion fires. Afterwards the ordering will remain static.

Table 1 details the coverage results of the three different versions of Gamer on the IPC 2011 benchmarks. All runs were performed on machines containing two eight-core Intel Xeon E5-2660 CPUs with 2.20 GHz and 64 GB RAM, running 16 planners with 4 GB memory limit and 30 minute timeout in parallel. Note that these machines differ from the one used at IPC 2011, and that we are restricting the planners to only 4 GB of RAM (the setting of IPC 2014), while in 2011 6 GB were allowed, so that the numbers are not comparable to the results of IPC 2011.

From that table we see that already the basic improvements and the switch over to using only bidirectional Dijkstra search bring a slight improvement. However, this improvement is surprisingly small: In preliminary runs with 6 GB memory limit the differences between the IPC 2011 and IPC 2014 versions of Gamer were much more pronounced. There, the basic improvements already brought an

Domain (#Tasks)	Gamer '11	Gamer '14	Dynamic Gamer '14
barman (20)	5	7	8
elevators (20)	18	18	18
floortile (20)	7	12	12
nomystery (20)	13	12	14
openstacks (20)	20	20	20
parcprinter (20)	8	6	6
parking (20)	0	0	0
pegsol (20)	17	17	17
scanalyzer (20)	8	9	9
sokoban (20)	16	11	13
tidybot (20)	0	5	8
transport (20)	7	7	9
visitall (20)	9	9	12
woodworking (20)	15	16	16
Total (280)	143	149	162

Table 1: Coverage results of different versions of Gamer on the set of IPC'11 benchmarks.

increase in coverage of 13, and the change to bidirectional Dijkstra another increase of 5 (as detailed in (Edelkamp, Kissmann, and Torralba 2012)). In our setting, which is the same as in IPC 2014, the change from Gamer '11 to Gamer '14 brings an increase in total coverage of only 6.

Considering dynamic reordering, the change here is much more pronounced than in previous runs with 6 GB of memory. In all domains, switching on dynamic reordering never hurts but only increases the number of solved instances, in total bringing us from 149 solutions up to 162. This indicates that dynamic reordering is especially helpful in cases of very limited memory.

Considering the change in the grounding utility, we also analyzed the binary encoding sizes of the resulting finite domain representations (cf. Table 2). From this we can see that in most domains we can do at least a bit better than with the IPC'11 version. The biggest advantage comes with Tidybot, where the positions of the base and the cart are now correctly identified as FD variables (here we have a situation where the coordinates are given by two arguments), which clearly brought an increase in coverage. On the other hand, the decrease in encoding size in Sokoban was counterproductive, as this results in an encoding of the different stones as FD variables instead of the different cells. While the encoding size decreases, this creates a number of additional binary variables (cell empty or not), overall decreasing coverage in this domain. However, as we noticed an increase in total coverage in further preliminary experiments we decided to keep the new grounding approach. The question of how to overcome the problem we noticed in Sokoban remains future work.

Conclusion

In this paper we presented a number of extensions to the existing symbolic planner Gamer over the version that was used in IPC 2011. One extension concerns the used search algorithm: While in 2011 Gamer varied between using (bidirectional) breadth-first search and symbolic A* search in conjunction with (partial) pattern databases, we now changed over to a setting where we use bidirectional symbolic Dijkstra search in all cases. Another extension concerns the use of dynamic variable reordering, which seems especially promising in the setting of IPC 2014, where the memory is more limited than in 2011.

Finally, we also presented a way to handle conditional effects by constructing transition relation BDDs that explicitly use them. This has the advantage that we do not have to compile all the conditional effects away and thus prevent an exponential (in the number of conditional effects) blowup.

References

Anderson, K.; Holte, R.; and Schaeffer, J. 2007. Partial pattern databases. In Miguel, I., and Ruml, W., eds., *Proceedings of the 7th International Symposium on Abstraction, Reformulation, and Approximation (SARA-07)*, volume 4612 of *Lecture Notes in Computer Science*, 20–34. Whistler, Canada: Springer-Verlag.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35(8):677–691.

Burch, J. R.; Clarke, E. M.; Long, D. E.; McMillan, K. L.; and Dill, D. L. 1994. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13(4):401–424.

Burch, J. R.; Clarke, E. M.; and Long, D. E. 1991. Symbolic model checking with partitioned transition relations. In Halaas, A., and Denyer, P. B., eds., *Proceedings of the International Conference on Very Large Scale Integration (VLSI-91)*, volume A-1 of *IFIP Transactions*, 49–58. Edinburgh, Scotland: North-Holland.

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In Biundo, S., and Fox, M., eds., *Recent Advances in AI Planning. 5th European Conference on Planning (ECP'99)*, Lecture Notes in Artificial Intelligence, 135–147. Durham, UK: Springer-Verlag.

Edelkamp, S., and Helmert, M. 2001. MIPS: The model checking integrated planning system. *AI Magazine* 22(3):67–71.

Edelkamp, S., and Kissmann, P. 2009. Optimal symbolic planning with action costs and preferences. In Boutilier, C., ed., *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1690–1695. Pasadena, California, USA: Morgan Kaufmann.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In Herzog, O., and Günter, A., eds., *Proceedings of the 22nd Annual German Conference on Advances in Artificial Intelligence (KI'98)*, volume 1504 of *Lecture Notes in Computer Science*, 81–92. Bremen, Germany: Springer.

Task	barman		elevators		floortile		nomystery		openstacks		parcprinter		parking		pegsol		scanalyzer		sokoban		tidybot		transport		visitall		woodworking	
	11	14	11	14	11	14	11	14	11	14	11	14	11	14	11	14	11	14	11	14	11	14	11	14	11	14	11	14
1	68	67	27	27	38	34	17	17	55	54	42	38	108	88	100	72	12	12	54	40	139	85	30	30	5	5	63	59
2	68	67	35	35	38	34	22	22	60	59	52	48	108	88	100	72	24	24	100	61	183	109	30	30	5	5	63	59
3	68	67	35	35	47	40	24	24	65	64	51	47	125	102	100	72	24	24	93	53	183	109	24	24	12	12	76	71
4	68	67	31	31	47	40	28	28	70	69	106	98	125	102	100	72	32	32	81	51	183	109	30	30	12	12	69	65
5	87	85	31	31	56	48	39	39	75	74	65	59	125	102	100	72	50	50	109	68	235	135	34	34	19	19	75	71
6	87	85	36	36	56	48	44	44	80	79	176	164	125	102	100	72	32	32	117	88	235	135	34	34	19	19	87	82
7	87	85	50	50	62	52	48	48	86	85	139	129	167	121	100	72	108	108	73	56	235	135	38	38	29	29	82	76
8	87	85	35	35	62	52	52	52	91	90	92	84	167	121	100	72	60	60	95	63	235	135	45	45	29	29	100	93
9	108	105	40	40	74	60	56	56	96	95	75	69	167	121	100	72	70	70	113	70	295	165	42	42	41	41	101	95
10	108	105	40	40	74	60	60	60	136	135	133	123	167	121	100	72	80	80	152	98	295	165	56	56	41	41	99	93
11	108	105	55	55	86	68	16	16	101	100	158	146	187	136	100	72	32	32	79	54	295	165	46	46	54	54	111	104
12	108	105	40	40	86	68	22	22	106	105	103	95	187	136	100	72	32	32	oom	oom	295	165	61	61	54	54	99	93
13	147	144	39	39	92	72	24	24	111	110	217	203	187	136	100	72	32	32	94	59	295	165	54	54	69	69	106	99
14	147	144	44	44	92	72	28	28	116	115	262	254	187	136	100	72	32	32	106	68	375	205	61	61	69	69	121	113
15	147	144	43	43	108	91	38	38	121	120	311	293	232	151	100	72	50	50	116	76	371	201	59	59	87	87	129	120
16	147	144	35	35	108	91	44	44	131	130	364	354	232	151	100	72	60	60	117	76	383	205	66	66	87	87	120	113
17	173	169	39	39	124	108	48	48	126	125	145	140	207	151	100	72	70	70	100	61	379	205	44	44	106	106	148	137
18	173	169	50	50	124	108	52	52	141	140	171	159	207	151	100	72	80	80	155	119	375	205	39	39	106	106	140	131
19	173	169	40	40	129	105	56	56	146	145	211	197	227	188	100	72	108	108	118	72	379	205	44	44	127	127	142	153
20	173	169	45	45	129	105	60	60	151	150	197	183	227	188	100	72	60	60	77	59	383	205	59	59	127	127	156	145

Table 2: Comparison of encoding sizes of the IPC'11 tasks, using the grounding utility of Gamer'11 and Gamer'14.

Edelkamp, S.; Kissmann, P.; and Torralba, A. 2012. Advances in BDD search: Filtering, partitioning, and bidirectionally blind. In *3rd ICAPS-Workshop on the International Planning Competition (WIPC-12)*.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In Ghallab, M.; Hertzberg, J.; and Traverso, P., eds., *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)*, 274–283. Toulouse, France: Morgan Kaufmann.

Gazen, B. C., and Knoblock, C. 1997. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In Steel, S., and Alami, R., eds., *Recent Advances in AI Planning. 4th European Conference on Planning (ECP'97)*, volume 1348 of *Lecture Notes in Artificial Intelligence*, 221–233. Toulouse, France: Springer-Verlag.

Goldberg, A. V., and Werneck, R. F. F. 2005. Computing point-to-point shortest paths from external memory. In Demetrescu, C.; Sedgewick, R.; and Tamassia, R., eds., *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments and the Second Workshop on Analytic Algorithmics and Combinatorics (ALENEX/ANALCO'05)*, 26–40. Vancouver, BC, Canada: SIAM.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In Chien, S.; Kambhampati, R.; and Knoblock, C., eds., *Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS-00)*, 140–149. Breckenridge, CO: AAAI Press, Menlo Park.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, 162–169. AAAI Press.

Helmert, M.; Röger, G.; Seipp, J.; Karpas, E.; Hoffmann, J.; Keyder, E.; Nissim, R.; Richter, S.; and Westphal, M. 2011. Fast Downward Stone Soup. In *IPC 2011 planner abstracts*, 38–45.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge & shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the Association for Computing Machinery*. Accepted.

Hoffmann, J.; Edelkamp, S.; Thíebaux, S.; Englert, R.; Liporace, F.; and Trüg, S. 2006. Engineering benchmarks for planning: the domains used in the deterministic part of IPC-4. *Journal of Artificial Intelligence Research* 26:453–541.

Jensen, R. M.; Bryant, R. E.; and Veloso, M. M. 2002. SetA*: An efficient BDD-based heuristic search algorithm. In Dechter, R.; Kearns, M.; and Sutton, R. S., eds., *Proceedings of the 18th National Conference of the American Association for Artificial Intelligence (AAAI-02)*, 668–673. Edmonton, AL, Canada: AAAI Press.

Kissmann, P., and Edelkamp, S. 2011. Improving cost-optimal domain-independent symbolic planning. In Burgard, W., and Roth, D., eds., *Proceedings of the 25th National Conference of the American Association for Artificial Intelligence (AAAI-11)*, 992–997. San Francisco, CA, USA: AAAI Press.

Kissmann, P., and Hoffmann, J. 2013. What's in it for my BDD? On causal graphs and variable orders in planning. In

Borrajo, D.; Fratini, S.; Kambhampati, S.; and Oddi, A., eds., *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*, 327–331. Rome, Italy: AAAI Press.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

Nebel, B. 2000. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research* 12:271–315.

Rudell, R. 1993. Dynamic variable ordering for ordered binary decision diagrams. In Lightner, M. R., and Jess, J. A. G., eds., *Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design (ICCAD-93)*, 42–47. Santa Clara, CA, USA: IEEE Computer Society.

Optimal Planning using Flow-Based Heuristics

Blai Bonet
Universidad Simón Bolívar
Caracas, Venezuela
bonet@ldc.usb.ve

Menkes van den Briel
NICTA & Australian National University
Canberra, Australia
menkes@nicta.com.au

Overview

We incorporate flow-based heuristics (Bonet and van den Briel 2014) into the Fast Downward planning system (Helmert 2006). Flow-based heuristics are defined by the solution of a linear programming (LP) problem that models state variables in the planning problem as network flows. The idea of modeling state variables as network flows has been explored in previous works (Vossen et al. 1999; van den Briel, Vossen, and Kambhampati 2005), but using the corresponding LP solution as heuristic is more recent (van den Briel et al. 2007; Bonet 2013; Bonet and van den Briel 2014).

Flow-based heuristics can be quite informative on some planning tasks as they are not bounded by the optimal delete-relaxation heuristic h^+. Their heuristic value, however, can be improved considerably by adding a variety of constraints (Pommerening et al. 2014). In our planner, we add constraints derived from action landmarks and from variable merges. While adding constraints can increase the heuristic estimate, it generally also increases the heuristic evaluation. Hence, we tradeoff increased heuristic quality with increased heuristic computation time.

Below we provide a brief description of our LP model and discuss the additional constraints derived from action landmarks and variable merges.

Linear Programming Model

We consider SAS$^+$ planning with non-negative action costs. A SAS$^+$ problem is a tuple $P = \langle V, A, s_\circ, s_\star, c\rangle$ where V is a set of variables, each variable $X \in V$ with finite domain D_X, A is a set of actions, $s_\circ$ is the initial state, $s_\star$ is a goal description, and $c : A \to \mathbb{N}$ are non-negative action costs.

An atom in SAS$^+$ planning is a literal of the form $X = x$ where X is a variable and $x \in D_X$ is a variable value. For atom p, Var(p) and Val(p) denote the variable and variable value of p.

A SAS$^+$ action is a triple $\langle \text{Pre}, \text{Post}, \text{Prev}\rangle$, with Pre the set of preconditions, Post the set of post conditions, and Prev the set of prevail conditions of the action. The domain transition graph (DTG) for a variable $X \in V$ is a labeled directed graph with nodes for each value in D_X and a labeled arc for each transition in X. Labels corresponds to actions in the planning problem and because actions can cause the same transitions, arcs may have multiple labels.

In flow-based heuristics the planning problem is represented as a set of interacting network flow problems. Each variable $X \in V$ corresponds to a separate network flow problem, where nodes correspond to the variable values $x \in D_X$ and arcs correspond to the transitions between these values. These network flow problems are loosely connected as actions may cause transitions in multiple variables. A flow for a planning task $P = \langle V, A, s_\circ, s_\star, c\rangle$ is a function $f : A \to \mathbb{R}^+$ mapping arc labels into non-negative real numbers. A plan π defines an integral flow f_π where $f_\pi(a)$ is the number of occurrences of a in π.

For each atom p we setup the inequality over flows $f : A \to \mathbb{R}^+$:

$$LB_p \leq \sum_{(x',a,x)\in T_X} f(a) - \sum_{(x,a,x')\in T_X} f(a) \leq UB_p \quad (1)$$

where T_X is the set of transitions for the DTG and LB_p and UB_p are lower and upper bounds for the flow on p. The lower and upper bounds can be determined by checking whether p is in the initial state and/or in the goal description.

Action Landmarks

Action landmarks, or landmarks, (Hoffmann, Porteous, and Sebastia 2004; Helmert and Domshlak 2009) are state dependent. That is, landmarks that apply to one state may not necessarily apply to another state. For this reason, constraints derived from landmarks are added and removed from the LP model each time we evaluate a state. We compute landmarks using the LM-cut method (Helmert and Domshlak 2009).

Given a collection $\mathcal{L} = \{L_i\}_{i=1}^n$ of landmarks, a landmark $L \in \mathcal{L}$ for state s says that any plan for s must contain at least one action in L. In other words, landmark L induces a constraint of the form $\sum_{a\in L} f(a) \geq 1$ on any flow f for P. Since the collection of landmarks depends on the state s, landmark constraints are added to the LP when calculating the heuristic for state s and removed afterwards.

Variable Merges

Variable merges (Helmert, Haslum, and Hoffmann 2007; van den Briel et al. 2007) combines two or more variables

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

into one "super" variable. Whenever we create a merge, we represent the new variable as a network flow problem just like all the other variables. The problem with merged variables is that they typically have many more values than the variables it is composed of. Therefore, representing the complete network flow problem of each merged variable can become quite costly. In order to control the size of the LP model we use the concept of dynamic merging (Bonet and van den Briel 2014). Dynamic merging allows us to be very selective with merging variables. It allows us to incrementally merge more and more variables and allows us to incrementally merge the variables.

Engineering Considerations

Our implementation of variable merging is different from the merge-and-shrink heuristic (Helmert, Haslum, and Hoffmann 2007). In merge-and-shrink, variables are literally merged away. For example, after merging variables X and Y into the variable XY, only one variable exists, which is the variable XY. Our implementation of variable merging is similar to van den Briel et al. (2007) where no variable is ever removed. Thus, after merging variables X and Y into XY, three variables remain, X, Y, and XY. There are advantageous and disadvantageous for both implementations. The reason for not removing variables X and Y is that they can still be used in other merges. For example, we may only be interested in the merged variables XY and XZ, but not in the variable XYZ, which would not be possible in the implementation the merge-and-shrink heuristic.

Dynamic merging can be tricky to implement. The main thing is to avoid coding the complete variable merge. With dynamic merging we can partially merge variables which prevents us from having to create the cross product of domains as is done when merging variables in the traditional way. This helps us to keep the size of the LP model a small as possible.

Merging Strategy

Deciding when to merge and what to merge are two important questions that determine the success or failure of a merge strategy. Here, we adopt a simple merge strategy that considers pairs of atoms such that one atom is the prevail condition and the other atom is the precondition of an action. At a higher level, we look for causal links in the causal graph (Helmert 2004) that are introduced by actions with a prevail condition in one variable $X \in V$ and an effect in another variable $Y \in V$.

Note that, some hierarchical planners can handle causalities quite well and simply incorporate them directly into the hierarchical structure. For example, in a classic logistics planning task, a hierarchical planner may first find a plan for each package, then use these plans to impose ordered conditions on the trucks, and then find a plan for each truck. By merging variables we, in effect, handle causalities by "removing" or "compiling-in" the causal links between variables into the new merged variables.

We recognize that our merge strategy is by no means comprehensive as we ignore many other causal links between the variables. We do believe, however, that there is ample opportunity to try other, more involved, merge strategies in future work.

Acknowledgements

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

Bonet, B., and van den Briel, M. 2014. Flow-based heuristics for optimal planning: Landmarks and merges. In *Proc. ICAPS*. Portsmouth, NH: AAAI Press. To appear.

Bonet, B. 2013. An admissible heuristic for SAS$^+$ planning obtained from the state equation. In *Proc. IJCAI*, 2268–2274.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. ICAPS*, 162–169. Thessaloniki, Greece: AAAI Press.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS*, 176–183. Providence, RI: AAAI Press.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proc. ICAPS*, 161–170. Whistler, Canada: AAAI Press.

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

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research* 22:215–278.

Pommerening, F.; Röger, G.; Helmert, M.; and Bonet, B. 2014. Lp-based heuristics for cost-optimal planning. In *Proc. ICAPS*. Portsmouth, NH: AAAI Press. To appear.

van den Briel, M.; Benton, J.; Kambhampati, S.; and Vossen, T. 2007. An LP-based heuristic for optimal planning. In *Proc. CP*, 651–665. Springer: LNCS 4741.

van den Briel, M.; Vossen, T.; and Kambhampati, S. 2005. Reviving integer programming approaches for ai planning: A branch-and-cut framework. In *Proc. ICAPS*, 310–319. AAAI Press.

Vossen, T.; Ball, M.; Lotem, A.; and Nau, D. 1999. On the use of integer programming models in AI planning. In *Proc. IJCAI*, 304–309. Stockholm, Sweden: AAAI Press.

h^{++} and h^{++}_{ce}
(IPC 2014 Planner Abstract)

P@trik Haslum
Australian National University & NICTA Optimisation Research Group
firstname.lastname@anu.edu.au

Introduction

h^{++} and h^{++}_{ce} are not planners. They are incremental lower-bounding procedures, based on repeatedly computing cost-optimal plans for a relaxation of the planning problem and strengthening the relaxation. If the relaxed plan is valid also for the real (unrelaxed) problem, it is an optimal plan. It is in this way that they can be used as cost-optimal planners.

The complete details of the h^{++} and h^{++}_{ce} procedures are presented in the following papers:

- Patrik Haslum. "Incremental Lower Bounds for Additive Cost Planning Problems". ICAPS 2012.
- Emil Keyder, Joerg Hoffmann and Patrik Haslum. "Semi-Relaxed Plan Heuristics". ICAPS 2012.
- Patrik Haslum. "Optimal Delete-Relaxed (and Semi-Relaxed) Planning with Conditional Effects". IJCAI 2013.
- Patrik Haslum, John Slaney and Sylvie Thiébaux. "Minimal Landmarks for Optimal Delete-Free Planning" ICAPS 2012.

This abstract presents only a brief overview of the approach, and notes the few points on which the IPC 2014 versions differ from the previously published descriptions.

Incremental Semi-Relaxation

The initial relaxation is the standard delete relaxation, P^+, of the original problem P. Strengthening is done by a semi-relaxing transformation of P into a new problem. Two transformations are used: P^C (in h^{++}) and P^C_{ce} (in h^{++}_{ce}), where C, a set of conjunctions of propositions in the input problem, is a parameter of both. The central property of the semi-relaxing transformations is that $h^\star(P^C) = h^\star(P^C_{\mathsf{ce}}) = h^\star(P)$ but $h^+(P^C) \geq h^+(P^C_{\mathsf{ce}}) \geq h^+(P)$. In other words, they preserve optimal plan cost, but, for some C, increase the optimal relaxed plan cost. For a sufficiently large set C, $h^+(P^C) = h^+(P^C_{\mathsf{ce}}) = h^\star(P)$. This is achieved by introducing new propositions that correspond to the selected conjunctions. To maintain the correspondence, actions in the problem have to be modified. The two transformations differ in how this is done. The P^C construction yields a STRIPS problem (assuming STRIPS input) but may be exponential in $|C|$. The P^C_{ce} construction instead outputs a problem with conditional effects. It can also grow exponentially, but only in the number of minimal effect conditions that cause the same action to simultaneously add and delete part of some conjunction $c \in C$. The P^C_{ce} construction is weaker, in the sense that $h^+(P^C) \geq h^+(P^C_{\mathsf{ce}})$ with strict inequality for some P and C.

The set of conjunctions considered in the semi-relaxing transformation is found incrementally. In each iteration, an optimal relaxed plan for the current problem is found. If this plan fails to solve the real (unrelaxed) problem, the failure is analysed to find a set of new conjunctions, termed "flaws", such that representing those in the next relaxation prevents the same plan from being found again. Flaw extraction is complete for the P^C construction, meaning that it will always generate a sufficient set of conjunctions. For the P^C_{ce} construction, there exist cases where a sufficent set of flaws may not be found. In this situation, h^{++}_{ce} terminates with failure.

The optimal relaxed plan is found using the iterative landmark algorithm, as described by Haslum, Slaney & Thiébaux (2012), with one modification: Optimal hitting set problems are formulated as integer programs and solved with the CPLEX IP solver, in place of the branch-and-bound algorithm described in that paper.

Handling of Conditional Effects

h^{++} compiles away conditional effects (using the standard exponential compilation) from the input problem. Since it uses the P^C construction, no new conditional effects are generated. Thus, it operates only with STRIPS problems.

Since h^{++}_{ce} uses the P^C_{ce} construction, which introduces conditional effects, it also keeps any conditional effects present in the input problem. An incremental compilation approach is used to compute h^+ with conditional effects. It relaxes the association between conditional effects and actions (allowing effects to "float") to yield a (delete-relaxed) STRIPS problem. An optimal plan for this problem is found, and scheduled into a plan for the delete-relaxed problem with conditional effects. If scheduling is not possible, due to cyclic dependencies, a subset of conflicting effects are compiled away (using the exponential compilation) and the process repeats.

Metis: Arming Fast Downward with Pruning and Incremental Computation

Yusra Alkhazraji
University of Freiburg, Germany
alkhazry@informatik.uni-freiburg.de

Michael Katz
IBM Haifa Research Labs, Israel
katzm@il.ibm.com

Robert Mattmüller
University of Freiburg, Germany
mattmuel@informatik.uni-freiburg.de

Florian Pommerening
University of Basel, Switzerland
florian.pommerening@unibas.ch

Alexander Shleyfman
Technion, Haifa, Israel
alesh@technion.ac.il

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

Introduction

Metis is a sequential optimal planner that implements three components on top of the Fast Downward planning system (Helmert 2006). The planner performs an A^* search using the following three major components:

- an admissible incremental LM-cut heuristic (Pommerening and Helmert 2013),
- a symmetry based pruning technique (Domshlak, Katz, and Shleyfman 2012), and
- a partial order reduction based pruning technique based on strong stubborn sets (Wehrle and Helmert 2012).

Each of those techniques was extended to support conditional effects. In addition, Metis features a flexible invocation of partial order reduction based pruning. In what follows, we describe each of these components in detail.

Background

We consider planning tasks $\Pi = \langle V, O, s_0, G, Cost\rangle$ captured by the standard SAS$^+$ formalism (Bäckström and Klein 1991; Bäckström and Nebel 1995) with operator costs, extended by conditional effects. In such a task, V is a set of finite-domain *state variables*, each with domain $\mathcal{D}(v)$. Each complete assignment to V is called a *state*, and $S = \prod_{v\in V}\mathcal{D}(v)$ is the *state space* of Π. The state s_0 is the *initial state* of Π. We sometime refer to a single variable assignment as to *fact*. Furthermore, the goal G is a partial assignment to V, where a state s is a *goal state*, iff $G \subseteq s$[1]. The set O is a finite set of *operators*. Each operator o is given by a pair $\langle \mathsf{pre}, \mathsf{effs}\rangle$. The *precondition* $\mathsf{pre}(o)$ is a partial assignment to V that defines when the operator is applicable. The set $\mathsf{effs}(o)$ is a set of *conditional effects* e, each given by a pair $\langle \mathsf{cond}, \mathsf{eff}\rangle$ of partial assignments to V called *conditions* and *effects*. The condition $\mathsf{cond}(e)$ defines when the conditional effect triggers. For a shorter presentation, we assume that eff assigns a value to exactly one variable. An effect that assigns a value to more variables can be split into multiple effects. Effects that do not assign a value at all can be safely removed. Finally, $Cost : O \to \mathbb{N}_0$ is a real-valued, non-negative *operator cost* function. Applying an applicable operator o in state s results in a state denoted by $s[o]$. The state $s[o]$ is obtained from s by applying all triggered conditional effects of o, setting the value of the state variable to the value in $\mathsf{eff}(e)$. State variables that do not appear in triggered effects receive their values from the state s. By the transition graph $\mathcal{T}_\Pi = \langle S, E\rangle$ of Π we refer to the edge-labeled digraph induced by Π over S: if $o \in O$ is applicable in state s, then $\mathcal{T}_\Pi$ contains an edge $(s, s[o]; o)$ from s to $s[o]$, labeled with o.

[1] We slightly abuse the notation here, treating (partial) assignments as sets of facts.

Heuristic

Metis uses a variant of the admissible LM-cut heuristic (Helmert and Domshlak 2009). In particular, we use the local incremental LM-cut heuristic, $h^{\text{iLM-cut}}_{\text{local}}$ (Pommerening and Helmert 2013) extended to support conditional effects. In the following, we provide a short rehash how the computation of standard LM-cut works, and afterwards discuss the two extensions.

The computation of standard LM-cut is done in *rounds*. Each round discovers a set of operators L such that every plan must contain at least one operator from L. Such a set is called a *disjunctive action landmark* but since we do not use any other kinds of landmarks, we will just use the term *landmark* in the following.

Each round of the LM-cut algorithm does the following steps:

1. Compute the $h^{\max}$ values (Bonet and Geffner 2001) of all variables. If the goal has an infinite $h^{\max}$ value, the task is unsolvable and the heuristic computation stops with a heuristic value of ∞. If the goal has an $h^{\max}$ value of 0, the algorithm stops with the current heuristic value (which is initialized as 0).
2. Define the *justification graph* as the graph $J = (F, E)$ with the set of facts F as nodes and a directed, weighted edge in E for every effect of every operator. The edge for effect e of operator o starts from a precondition of o with maximal $h^{\max}$ value, ends in the single fact in $\mathsf{eff}(e)$ and is labeled with o and weighted with $Cost(o)$. All nodes in the justification graph that have a path to the goal where all edges have weight 0 belong to the *goal zone* $F_g \subseteq F$.

The *cut* C contains all edges that end in F_g and start in a node that can be reached in J without traversing a node in F_g. The set of operators that occur as labels of edges in C is a landmark L of the task.

3. The *cost* of L, $Cost(L)$, is the minimum over the cost of all operators contained in L. This reflects that at least the cost of one operator in L must be used. Reduce the cost of each operator in L by $Cost(L)$ and increase the heuristic value by $Cost(L)$. This induces a cost partitioning and makes the final estimate admissible.
4. Discard L.

After the last round, all operator costs are reset to their original value.

Support for Conditional Effects

The original LM-cut algorithm is only defined for tasks without conditional effects. We extended its definition to handle conditional effects by considering them in the definition of the justification graph in step 2. and conservatively reducing the operator costs in step 3. (Keyder, Hoffmann, and Haslum 2012). Following the naming convention of Röger, Pommerening, and Helmert (2014), we call this heuristic $h^{\text{LM-cut}}_{\text{basic}}$.

Our extended definition of the justification graph handles unconditional effects as before, and includes an edge for every conditional effect. The edge for an effect e of an operator o ends in the single fact in $\mathsf{eff}(e)$ and starts in a fact from $\mathsf{pre}(o) \cup \mathsf{cond}(e)$ with maximal $h^{\max}$ value. It is labeled with o and weighted with $Cost(o)$. The cut C and the landmark L are defined as before.

We call the reduction in cost conservative because the cost of each operator can only be counted once. Once an operator o is part of a cut, the cost of o is reduced, and all effects of o are cheaper. In the presence of conditional effects the optimal relaxed plan can contain operators more than once which our heuristic would not be able to detect. For this reason our heuristic no longer dominates the $h^{\max}$ heuristic (Keyder, Hoffmann, and Haslum 2012). Röger, Pommerening, and Helmert (2014) describe a variant of LM-cut that dominates $h^{\max}$ but this is not implemented in Metis.

The original LM-cut heuristic (without support for conditional effects) $h^{\text{LM-cut}}_{\text{standard}}$ is the same as $h^{\text{LM-cut}}_{\text{basic}}$ on tasks without conditional effects but can be implemented more efficiently because the involved data structures have less memory and time overhead. Metis thus includes both implementations and uses $h^{\text{LM-cut}}_{\text{basic}}$ only on tasks where at least one operator has a conditional effect.

Incremental Computation

A set of operators L is a landmark for state s if every plan from s must use one operator from L. If we apply an operator $o \notin L$ to s, the resulting state s' can only have plans that are suffixes of plans for s. That is, if we add o in front of any plan π for s', we get a plan for s. Since every plan for s must use an operator from L and $o \notin L$, every plan for s' must also use an operator from L and L is also a landmark for s'.

In particular, this means that during the expansion of a state all landmarks that do not mention the applied operator are also landmarks of the successor. If we know landmarks $\mathcal{L}$ of the parent state when we calculate the heuristic value for a newly generated state, we can compute the set of landmarks $\mathcal{L}' = \{L \in \mathcal{L} \mid o \notin L\}$ of all landmarks that do not mention the applied operator o. For each $L \in \mathcal{L}'$ we then reduce the operator costs and increase the heuristic value as defined in step 3. and continue with the regular LM-cut algorithm.

Storing the discovered landmarks for all generated states, can make the heuristic computation much faster but also requires a lot of memory (Pommerening and Helmert 2013). Instead, we use the local incremental computation method $h^{\text{iLM-cut}}_{\text{local}}$ which recomputes the landmarks of a state before it is expanded. This is done with a regular (non-incremental) LM-cut computation that skips step 4. The landmarks are then stored temporarily, used for the incremental heuristic computation of the generated children and are then discarded. With this method the search will do one non-incremental heuristic computation for every expanded state and one incremental computation for every generated state. Since there usually are a lot more generated than expanded states and the incremental computation is faster, the time for the additional non-incremental computation can be amortized.

Symmetry Reduction

The symmetry pruning part of the Metis planner modifies the A^* algorithm to prune symmetrical search nodes. For that, we needed to (a) develop a mechanism identifying symmetrical states, and (b) exploit the information in the search. In what follows, we describe these two in detail.

Symmetries and Conditional Effects

In what follows, we discuss the symmetries of of the state transition graph $\mathcal{T}_\Pi$ of a SAS^+ planning task Π that are captured by automorphisms (isomorphisms to itself) of $\mathcal{T}_\Pi$. As the state transition graph $\mathcal{T}_\Pi$ is not (and cannot be) given explicitly, automorphisms of $\mathcal{T}_\Pi$ must be inferred from the description of Π. The specific method that Pochter, Zohar, and Rosenschein (2011) proposed for deducing automorphisms of $\mathcal{T}_\Pi$ exploits automorphisms of a certain graphical structure (colored graph), the *problem description graph (PDG)*, induced by the description of Π. Later, Domshlak, Katz, and Shleyfman (2012) slightly modified the definition, in particular extending it with a support for non-uniform cost operators. Here we extend the definition of Domshlak et al. to support conditional effects.

In the regular SAS^+ setting, the PDG has one node for each operator, with incoming edges from variable values in the operator precondition, and outgoing edges to variable values in the effect. The operator nodes are colored according to their costs. When conditional effects come into the picture, an additional node is introduced for each conditional effect. The edges for such nodes are as follows. An incoming edge is added from each variable value in the effect's condition. An outgoing edge is added to the variable value

in the effect. In addition, to preserve the connection between the conditional effect to its operator, an incoming edge is added from the operator node. The color of the conditional effect nodes is the same as the color of the corresponding operator node.

Automorphisms of the PDG define isomorphisms on the states S of the planning task Π, such that if a state s is mapped into s', then s and s' are symmetrical. Given a set of automorphisms of the PDG, Pochter et al. define a procedure, mapping each state to a *canonical* symmetrical state. Obviously, there can be multiple ways to define canonical states. Pochter et al. have chosen a local search procedure, comparing states by their variable values. The procedure terminates with a local minimum. Our implementation adopts their approach with a minor modification to the local search procedure.

Search Algorithm

In order to exploit the information about the problem's symmetries, Domshlak, Katz, and Shleyfman (2012) propose a sound and complete optimal search algorithm (hereafter referred to as DKS), extending A^* as follows. First, DKS extends the duplicate elimination mechanism to consider symmetrical states as duplicates. To do so, DKS requires storing an additional information for each node – the canonical state. Two states are then said to be duplicates, if their canonical state is the same. Unfortunately, using such duplicate elimination comes at a certain cost. When reopening is required, the parent relation, if updated, loses the connectivity property. Thus, once a goal state is reached, it is no longer possible to retrace a path from the goal state by the parent relation. Therefore, DKS introduced a procedure exploiting the symmetry information for reconstructing a plan following the parent relation, without requiring its connectivity.

To overcome the aforementioned requirement of storing an additional state per node for duplicate elimination, we introduce a simple modification of the DKS search algorithm. It is called *orbit search*, and it differs from DKS by storing *only* the canonical state per node. These canonical states define *orbits*, sets of (symmetrical) states that have the same canonical state – thus the name orbit search. Informally, orbit search searches in the space of orbits instead of the space of states. Note that due to the plan reconstruction procedure of DKS, the implementation of orbit search is extremely simple. The syntactical difference between A^* and orbit search is minor, the states are replaced by their canonical representatives when stored in the open and close lists. Our preliminary experiments have shown an advantage of orbit search over DKS, and the less complex implementation makes it especially attractive.

Partial Order Reduction

In addition to symmetry pruning, Metis features a pruning technique based on strong stubborn sets for planning (Wehrle and Helmert 2012), which is a state-based pruning technique based on partial order reduction (POR). In a nutshell, POR attempts to reduce the size of the reachable state space by pruning redundant applications of operator sequences. In the following, we describe strong stubborn sets, and their extension to support operators with conditional effects.

Strong Stubborn Sets

Let Π be a SAS$^+$ planning task without conditional effects. For a state s in Π, a strong stubborn set T_s in s is a set of operators that satisfy the following three requirements: First, T_s contains the operators of a disjunctive action landmark. Second, for all operators o in T_s that are applicable in s, T_s contains all operators that *interfere* with o. Informally, operators o and o' interfere if o falsifies a precondition of o', or vice versa, or o and o' write to a common variable with different values (see below for a definition in presence of operators with conditional effects). Third, for all non-applicable operators o in T_s, T_s contains a *necessary enabling set* for o in T_s. A necessary enabling set N for an inapplicable operator o in s is a set of operators such that every plan π_s from s to a goal state that includes o must include an operator from N before the first occurrence of o in π_s. We compute strong stubborn sets T_s in s with a fixed-point iteration. Generating successors only based on the applicable operators in T_s (instead of all operators applicable in s) preserves completeness and optimality of A^*.

Metis features a rather straight-forward implementation of strong stubborn sets, including some optimizations to reduce the induced computational overhead.

- Previous implementations of strong stubborn sets compute the interference relation between operators in a preprocessing step and cache the result (Alkhazraji et al. 2012; Wehrle et al. 2013). However, in domains with many operators, the precomputation can run out of memory due to the quadratic number of operator pairs.

  Metis computes the relation for operator interferences ("which pairs of operators interfere?") and achievers ("which fact is added by which operator?") on-the-fly and caches the result until a limit of 100 million entries in total is exceeded. In this case, we stop caching, and compute the missing information on-the-fly without storing it.
- In order to avoid unnecessary computational overhead induced by the fixed-point iteration for computing strong stubborn sets, we switch off this computation if we do not get significant pruning. In more detail, if it turns out that the number of node generations is reduced by less than one percent compared to not using POR after at least 1000 node expansions, then the computation of strong stubborn sets is disabled for the rest of the search on this task.
- Necessary enabling sets for o and s are computed by selecting a precondition fact f of o that is unsatisfied in s, and including all operators that set f to true. Metis uses a straight-forward instantiation by greedily selecting the first unsatisfied precondition fact of o. This strategy has been proposed by Alkhazraji et al. (2012), and corresponds to the static *Fast Downward* ordering investigated by Wehrle and Helmert (2014). Analogously, the disjunctive action landmark used to start the fixed-point iteration

to compute strong stubborn sets is obtained by selecting all achievers of an unsatisfied goal fact.

In the following, we describe the extension of the above implementation to deal with conditional effects.

Support for Conditional Effects

Metis treats operators with conditional effects in a conservative (and straight-forward) way based on the following modifications.

- Definition of *operator interference*: Operator $o = \langle \mathsf{pre}, \mathsf{effs} \rangle$ *disables* operator $o' = \langle \mathsf{pre}', \mathsf{effs}' \rangle$ iff there is at least one conditional effect $e = \langle \mathsf{cond}, \mathsf{eff} \rangle \in \mathsf{effs}(o)$ such that eff falsifies a precondition fact in pre' or a fact in the effect condition cond' for a conditional effect $\langle \mathsf{cond}', \mathsf{eff}' \rangle \in \mathsf{effs}(o')$. In other words, the conditions of the conditional effects are handled exactly as preconditions. Operators o and o' have *conflicting effects* iff there is a conditional effect $e = \langle \mathsf{cond}, \mathsf{eff} \rangle \in \mathsf{effs}(o)$ and a conditional effect $e' = \langle \mathsf{cond}', \mathsf{eff}' \rangle \in \mathsf{effs}(o')$ such that eff and eff' modify the same variable with a different value. Operators o and o' *interfere* if o disables o', or vice versa, or o and o' have conflicting effects.
- During the iterative computation of a strong stubborn set T_s in state s, for all operators $o = \langle \mathsf{pre}, \mathsf{effs} \rangle$ applicable in s, we add all operators that interfere with o as for SAS^+ planning tasks. In addition, for conditional effects $e = \langle \mathsf{cond}, \mathsf{eff} \rangle \in \mathsf{effs}(o)$ with unsatisfied effect condition cond in s, the set of achievers for an unsatisfied fact of cond is added. Adding such sets is required for preserving the soundness of the algorithm. Intuitively, such sets correspond to necessary enabling sets for non-applicable operators as compiling away conditional effects would result in corresponding non-applicable operators.

Finally, in domains that do not feature conditional effects, Metis additionally performs pruning based on active operators (Wehrle et al. 2013).

Interaction of Components

The partial order reduction technique is orthogonal to the other two techniques. There are no special considerations to consider when combining it with either one or both of them.

However, combining the symmetry based pruning technique with the incremental computation of the heuristic function requires some extra considerations. If a node with state s is expanded in orbit search, the state s' of the successor generated for operator o is not necessarily the result of applying o to s. Orbit search instead uses the canonical representative of $s[o]$ as the state s'. Calculating landmarks for s and re-using those that do not mention o as landmarks for s' thus is no longer valid.

We handle this issue in the following way: during the expansion of state s, we compute the landmarks for s, then generate the actual successor $s[o]$ and incrementally compute its heuristic value. We then look up s', the canonical representative of $s[o]$. Because of the symmetry between the two states, the heuristic value of $s[o]$ can also be used as an admissible estimate for s'. The search algorithm generates the successor node with the state s' but the heuristic value $h^{\text{iLM-cut}}_{\text{local}}(s[o])$. The state $s[o]$ is not stored permanently to save memory.

Acknowledgments

This work was partly supported by the German Research Foundation (DFG) with grant HE 5919/2-1 and by the Swiss National Science Foundation (SNSF) as part of the project "Safe Pruning in Optimal State-Space Search (SPOSSS)".

References

Alkhazraji, Y.; Wehrle, M.; Mattmüller, R.; and Helmert, M. 2012. A stubborn set algorithm for optimal planning. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)*, 891–892.

Bäckström, C., and Klein, I. 1991. Planning in polynomial time: The SAS-PUBS class. *Computational Intelligence* 7(3):181–197.

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

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.

Domshlak, C.; Katz, M.; and Shleyfman, A. 2012. Enhanced symmetry breaking in cost-optimal planning as forward search. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS).*

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: Whats the difference anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS).*

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

Keyder, E.; Hoffmann, J.; and Haslum, P. 2012. Semi-relaxed plan heuristics. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 128–136.

Pochter, N.; Zohar, A.; and Rosenschein, J. S. 2011. Exploiting problem symmetries in state-based planners. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI).*

Pommerening, F., and Helmert, M. 2013. Incremental lm-cut. In *Proceedings of the 23nd International Conference on Automated Planning and Scheduling (ICAPS)*, 31–41.

Röger, G.; Pommerening, F.; and Helmert, M. 2014. Optimal planning in the presence of conditional effects: Extending LM-Cut with context splitting. In *ICAPS 2014 Workshop on Heuristics for Domain-Independent Planning*.

Wehrle, M., and Helmert, M. 2012. About partial order reduction in planning and computer aided verification. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 297–305.

Wehrle, M., and Helmert, M. 2014. Efficient stubborn sets: Generalized algorithms and selection strategies. In *Pro-*

ceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS). To appear.

Wehrle, M.; Helmert, M.; Alkhazraji, Y.; and Mattmüller, R. 2013. The relative pruning power of strong stubborn sets and expansion core. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS)*, 251–259.

RIDA:*In Situ* Selection of Heuristic Subsets

Santiago Franco and **Mike Barley** and **Pat Riddle**
Computer Science Department
Auckland University
Auckland, New Zealand

Abstract

Progress has been made in developing techniques to automatically generate effective heuristics. These techniques aim to reduce the size of the search tree, usually by combining more primitive heuristics. However, simply reducing search tree size is not enough to guarantee that problems will be solved more quickly. In this paper we summarize the planner RIDA*, a new approach to automatic heuristic generation that combines more primitive heuristics in a way that can produce better heuristics than current methods. A more complete description can be found in (Barley, Franco, and Riddle 2014).

Introduction

In this paper we briefly describe the planning system called RIDA*[1] Most of the information given in this paper is taken from (Barley, Franco, and Riddle 2014).

In the last few years, multiple techniques (Haslum et al. 2007; Haslum, Bonet, and Geffner 2005; Edelkamp 2007; Nissim, Hoffmann, and Helmert 2011; Helmert, Haslum, and Hoffmann 2007; Pommerening and Helmert 2012; Pommerening, Röger, and Helmert 2013) have been developed to automatically generate heuristics from domain and problem specifications. In this paper, we call the components that generate these heuristics *heuristic generators*. Frequently, different heuristic generators perform better on different domains. In some cases, combining different heuristic generators can result in either solving more problems or reducing the overall solving times.

RIDA* chooses heuristics by reasoning about how those choices will impact the problem-solver's search time. RIDA* is able to reason about a heuristic's impact on search time because it measures both per node generation time and per node heuristic evaluation time. RIDA* then uses a runtime formula to predict the impact of a heuristic upon search time by using its estimates of the heuristic's impact on the average time per node and on the tree size. In this way RIDA* can base its choice of heuristics upon their predicted impact on search time.

RIDA* is implemented on top of Fast Downward. For the planning competition we are using the following PDB-based heuristics: GA-PDBs(Edelkamp 2007),PDB-LP(Pommerening, Röger, and Helmert 2013), iPDB(Haslum et al. 2007) as implemented in Fast Downward. We are also using Incremental-lmcut(Pommerening and Helmert 2012) and blind (brute force A*) search. Surprisingly, for domains in which the heuristics do not do well having the option of blind search can solve a few more problems. Finally, when dealing with conditional effect domains, we use a different heuristic set including just h^{max}(Bonet and Geffner 2001) and blind. These are the only two heuristics available to us which can deal with conditional effects. We are aware that conditional effects can be "compiled away" but this frequently results in unwieldy computational increases, so we decided to stick to those heuristics which can deal with conditional effects.

Planner Description

Our approach's focus is on reducing the overall system's time to solve the problem. RIDA* trades the quality of the selected combination against the cost of making that selection. We try to reduce this selection cost primarily by reducing the time to predict each combination's effective branching factor.

RIDA* predicts the expected time to fully expand the current f-boundary for up to 20,000 different heuristic combinations. In our experiments, the current implementation of RIDA*'s heuristic generation process takes almost 70% of the total problem solving time. The generation time includes sampling time and utility calculation time.

We will first look at the different phases that RIDA* uses in solving a problem, then at our utility formula, next at the combination pruning heuristics used to reduce the combinatorics of the selection process and finally at the mechanisms that underlie our approach to reducing the time to approximate the combinations' effective branching factors (EBFs): the min combiner and culprit counters.

RIDA* Phases RIDA* inputs a set of heuristics and heuristic generators and outputs a heuristic. To do this it goes through a sequence of phases (which are described in

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1](Barley, Franco, and Riddle 2014) referred to the planning system as RA*, but for the planning competition its name was changed to RIDA* because order planners had similar names.

the next few sections):

- Run heuristic generators to produce "primitive" heuristics.
- Time how long it takes to generate a node.
- Time how long each primitive heuristic takes to evaluate a state.
- Initial Growth Period: Expand the initial part of the problem's search tree, maximizing available heuristics, to an adequate frontier size to start sampling.
- Sample Period: Nodes are taken from the frontier to obtain information on each candidate combination's EBFs.
- Estimate the utility for each candidate heuristic combination.
- Finish solving the problem with the best combination.

Utility Formula The utility formula for a combination c is:

$$U_{c,f} = \mid Front_f \mid * EBF_{c,f} * (eRN_c + gRN) \quad (1)$$

where f is the current f-level, $Front_f$ is the set of frontier nodes at the beginning of f, $EBF_{c,f}$ is the effective branching factor at f for combination c, eRN_c is the per node evaluation time, and gRN is the per node generation time. We obtain $EBF_{c,f}$ from sampling this f-level as described in the rest of this section. $EBF_{c,f}$ is all the sampled nodes by c in that f-level divided by the number of sampled frontier nodes. eRN_c is the sum of the evaluation times of all the heuristics in the combination.

Pruning Heuristics For any non-trivial set of heuristics, the number of combinations is large. In our experiments, we use a set of 45 heuristics. Thus there are 2^{45} (approximately $32 * 10^{12}$ combinations). Far too many to predict times for. Instead, we want to eliminate as many of the clearly inferior combinations as possible. To do this we use some rules of thumb to identify and prune away less useful primitive heuristics. We now discuss the main rules.

At the end of the initial growth period, we randomly select frontier nodes[2] to grow during the sampling period. We call these nodes the *sample roots*. We classify the primitive heuristics into strong, medium, and weak by evaluating every sample root using every heuristic. We count the times each heuristic had the highest value for a sample root. If a heuristic's count is high enough[3], it is called a *strong* heuristic. Of the remaining heuristics, with associated non-zero counts, the top user-specified percentage[4]are called *medium* heuristics. The remaining ones are called *weak*. A *combination candidate* is defined to be one which has at least one strong primitive heuristic, and the remainder are at least medium primitive heuristics. We generate the set of combination candidates in the ascending subset size order and prune away the ones that do not meet our criteria (specified above). We stop generating after a user-specified number[5], *MaxComb*, of candidate combinations.

Min Combiner and the Search Tree We grow only one search tree, a *MinTree*, rather than a separate tree for every combination. We do this by using a *min* combiner. A min combiner is very much like a max combiner, except that the heuristic value of a min combiner over a set of heuristics is the minimum value produced by that set of heuristics.

We need to grow the search tree deep enough, that our approximations will be reasonable. We call this period, the ***initial growth period*** . While we are in the initial growth period, we use the max combiner to keep the tree size small. When the tree is deep enough, at the end of an f-boundary, we switch to using the min combiner for one f-level. We call this f-level the ***sampling period***. Using the max combiner, a node is expanded only if all the heuristics agree to expand it. However, for the min combiner, a node is expanded as long as one of the heuristics wants to expand it.

Culprit Counters and Combination Counters In this section we give a very brief overview of how RIDA* computes the number of nodes generated by each heuristic combination during the sampling period. Franco et al. (2013) provide a more detailed description of this process.

During the sampling period, culprit counters are used to compute the effective branching factors for each combination. The min combiner allows us to only grow one search tree and the culprit counters allow us to only update one counter (the culprit counter) per node expanded.

When we expand a node, the primitive heuristics that agreed to the expansion are call the *culprits*. The culprits are a subset of the full set of heuristics. We associate a counter with each culprit set[6], the *culprit counter*. When a node is expanded in the sampling period, we add its number of children to the counter associated with its culprit set.

When the sampling period is over, we now need to calculate the number of nodes generated for each combination. During the final search to find a solution, RIDA* will be using the max combiner, which means that all the heuristics in the combination must agree to generate a node for it to be generated. This means that for each combination of heuristics, we compute how many nodes would have been generated for this f-level. For a given combination, we sum all the culprit counters whose culprit set heuristics are a subset of the combination's heuristics. After the counts for all the combinations have been summed, we can compute the effective branching factor for each combination by dividing its count by the number of sampled roots. We can also calculate the per node evaluation time for that combination of heuristics by adding together the per node evaluation time of each heuristic in that combination.

Once, the effective branching factors and the per node evaluation times have been calculated, then the utility for each combination is calculated and the combination with the highest utility is selected to finish the search for the solution.

[2]In our experiments RIDA* selects 1% of the previous f-level to sample.

[3]In our experiments, the count must be more than half of the number of samples taken.

[4]In our experiments, the user-specified percentage is 30%.

[5]In our experiments, this was set to 20,000

[6]We only store counters for the culprit sets actually encountered during the sampling period.

Current work

We have described the planner as in Barley et al (2014), in the mean time we have made the following modifications.

First, we realized that our selection method is biased towards more accurate heuristics. Lets say that we have two heuristics, h1 and h2, and h1 is more accurate. Furthermore, lets say that for one of the selected nodes in the sampling phase the f-value of h2 (*f2*) is less than the f-value of h1. In that case, the EBF we are calculating for h2 is incorrectly assessed. This is due to the fact that for some of the randomly selected sampled nodes, the lower accuracy heuristics give a lower f-value. This results in RIDA* calculating the EBF for h2 based on a bigger f-boundary transition. For h1, we only sampled the last f-boundary transition. In order for RIDA*'s selection to be unbiased, we need to ensure that only a common f-boundary transition is sampled for all heuristics. We have developed a mechanism, called subtract counters, to eliminate this bias. The subtract counters have the same structure as the culprit counters, but account for every time a node is expanded bellow the currently sampled f-level. When calculating the number of nodes generated for each heuristic combination, the corresponding subtract counters are subtracted to eliminate any nodes whose f-value is bellow the currently sampled f-level.

Secondly, we are dropping poor performing heuristics more aggressively. If a heuristic's f-value for the initial state is below another heuristic we check whether it is significantly more expensive (currently 10 times or more). If it is, then we eliminate the more expensive heuristic. It is very unlikely for such a heuristic to be part of RIDA*'s selected combination.

Finally, RIDA* does not take into account memory costs. This means that RIDA* can make a bad choice memory-wise, i.e. the combination is the fastest but also runs out of memory before other combinations which do find the solution before the memory limit is reached. The best way to deal with this is to take memory costs into account in the utility formula. However, this is ongoing work. For the competition we added two rules of thumb. First, if adding a heuristic to the selected combination reduces the number of generated nodes while only increasing slightly (currently up to 20%) the overall expected solving time, we add the heuristic. This rule of thumb aims to bias RIDA*'s selections towards heuristic combinations with smaller memory footprints, as long as the increase in solution time is reasonable. If we still run out of memory, we use a second rule of thumb. We rerun the search but select all the strong heuristics. Once again, this is far from ideal, but it can help when RIDA* quickly runs out of memory while it has only used a small portion of the allocated time to solve the problem.

IPC 2011 Experiments

In (Barley, Franco, and Riddle 2014) we compare the performance of RIDA* with both the primitive heuristics and also the standard maximization/randomization combination methods. Here we reproduce in Table 1 the results in terms of number of problems solved under the ICP 2011 time and memory limits. The list of primitive heuristics is missing the PDB_LP and Incremental_LmCut heuristics, which were not available at the time. The problems suite is the STRIPS optimal ICP2011 suite.

Looking at Table 1 we see that RIDA* was able to solve 187 of the 280 problems. This was better than the other systems. RIDA*'s heuristics failed to solve 5 of the 192 solved by at least one of the systems. In each of these 5 cases, RIDA*'s heuristic caused A* to exceed the memory limit rather than the time limit. A clear improvement to RIDA* would be to take the memory costs into account as well. This is future work.

Finally, Table 2 shows that even though RIDA* solves more problems, it is not the fastest method. The main reason for this is that RIDA* uses most of its time (in average 70%) generating all of the alternative PDBs, while the other methods only generate a subset of them. More detailed experimental data can be found in Barley et al (2014).

	Avg.	Std. Dev.	Sum
GA-D	205.79	371.40	38,700
iPDB	227.68	469.94	43,700
GA-ND	232.68	333.67	44,700
RIDA*	284.97	319.38	54,700
LM-cut	446.62	699.15	85,800
MAX	453.31	571.95	87,000
RAND	517.68	607.15	99,400
h^{max}	706.12	820.01	136,000

Table 2: The Total Times for Solving Problems

Conclusions

Our main claim is that heuristic generators can suffer from the utility problem. The utility problem is when the generator's attempts to create heuristics that generate smaller search trees also makes the system take longer to solve the problem.

Creating better heuristics involves reasoning about the heuristics' impact upon search time. Smaller search trees do not guarantee shorter search times. What is necessary, however, is for the heuristic combiner to make the best tradeoffs between the heuristic's reductions in the search tree size and any additional per node evaluation costs. RIDA* is a system that explicitly reasons about the heuristics' impact on the system's search time. The experiments presented in this paper were done in the 2011 IPC setting. These experiments use a set of five state-of-the-art[7] admissible heuristics and compare RIDA*'s performance with two default approaches (Max and Random). RIDA* is clearly superior to both approaches. These experiments also compare RIDA*'s performance against the individual heuristics. RIDA*'s heuristics, on average, reduce the search time more than any of the individual approaches. More extensive analysis in (Barley, Franco, and Riddle 2014) of the experiments indirectly support our claim that RIDA*'s superiority comes from its explicit reasoning about the heuristics' impact on search time.

[7]This was true on 2012, when out ICAPS paper experiments were performed. For the IPC 2014 competitions we have added PDB_LP and substituted LmCut with Incremental_LmCut. We are looking forward to review the results in the IPC 2014 domains.

	Problems Solved RIDA* problems solved (problems solved after sampling phase)									Avg. # Heuristics Used By RIDA*		
	RIDA*	GA-ND	GA-D	MAX	iPDB	LM-cut	RAND	h^{max}	total	GA	iPDB	LM-cut
Barman	4(4)	4	4	4	4	4	4	4	4	1.333	0.25	1
Elevators	19(9)	19	19	18	16	18	16	13	19	2.22	0	0
Floortile	6(4)	3	4	6	2	6	2	6	6	3.5	0	0.5
Nomystery	20(4)	20	20	20	18	14	20	8	20	3.25	0	0
Openstack	15(10)	16	16	13	16	16	15	16	16	1	0	0
Parcprinter	13(3)	13	13	13	11	13	10	11	13	1.67	0	0.33
Parking	7(7)	1	1	1	7	1	0	0	7	0	1	0
Pegsol	19(14)	19	19	17	18	17	17	17	19	6.86	0	0
Scanalyzer	14(4)	10	9	11	10	11	6	6	14	5	0	0
Sokoban	20(9)	20	20	20	20	20	20	20	20	0.67	0.67	0
Tidybot	13(11)	12	11	11	13	12	7	5	13	0.09	1	0
Transport	9(4)	10	10	8	6	6	7	6	10	1.25	0	0
Visitall	18(2)	16	17	17	16	10	13	9	18	7	0	0
Woodworking	10(5)	10	9	11	7	11	5	4	13	1.8	0	0
Total	187(92)	173	172	170	164	159	143	123	192	2.9	weighted average	
Ratio	.97(.49)	.89	.88	.88	.85	.83	.74	.64				

Table 1: Number of Problems Solved by Each System

While RIDA* generates better heuristic combinations, how does it compare to other systems that generate heuristics? In this paper we used the 2011 IPC criteria. The 2011 IPC criteria for deterministic optimal planners was the total number of problems solved under given time and memory constraints. RIDA* did best with 187 problems solved out of 280, while the next best system, GA-ND, solved 173 problems. However, when we compared RIDA*'s total runtime against the seven other systems, it came in 4th. This is unsurprising as RIDA* generates all 43 PDBs (using GA-D, iPDB, and GA-ND), while the top 3 systems (GA-D, iPDB, and GA-ND) only generated 21, 1, and 21 PDBs, respectively. The PDB generation time took up 70% of RIDA*'s total time.

The greatest improvement for RIDA* would likely come from better handling of PDB generation. Currently, all the candidate PDBs are generated and then they are evaluated at one time. It seems plausible that a better approach is to adopt the approaches of iPDB and GA-PDB to integrate the generation and evaluation of heuristics into an incremental search. This integration would have the drawback of making it more difficult to add new heuristic generators to RA*.

References

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

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1):5–33.

Edelkamp, S. 2007. Automated creation of pattern database search heuristics. In *Model Checking and Artificial Intelligence*, volume 4428 of *LNCS*. 35–50.

Franco, S.; Barley, M.; and Riddle, P. 2013. A new efficient in situ sampling model for heuristic selection in optimal search. In *Australasian Joint Conference on Artificial Intelligence*.

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.; Bonet, B.; and Geffner, H. 2005. New admissible heuristics for domain-independent planning. In *AAAI*.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, 176–183.

Nissim, R.; Hoffmann, J.; and Helmert, M. 2011. Computing perfect heuristics in polynomial time: On bisimulation and merge-and-shrink abstraction in optimal planning. In *IJCAI*, volume 3, 1983–1990.

Pommerening, F., and Helmert, M. 2012. Optimal planning for delete-free tasks with incremental lm-cut. In *ICAPS*.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the most out of pattern databases for classical planning.

Acknowledgments

This material is based on research sponsored by the Air Force Research Laboratory, under agreement number FA2386-12-1-4018. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.

We would also like to thank Malte Helmert, et al, for making available the Fast Downward code repository and the 2011 International Planning Competition domains and problems. Both of these have made doing research a lot easier. We would also like to thank Florian Pommerening for his generous contribution of two of the latest state-of-the-art heuristic generators (PDB_LP) and Incremental_LmCut. Finally also thanks to both Malte Helmert and Florian Pommerening for their generous and prompt technical guidance whenever we had doubts on how FD worked.

The Rational Lazy A* Planner

Erez Karpas
CSAIL
MIT

David Tolpin
Tal Beja
Solomon Eyal Shimony
CS Department
Ben-Gurion University
Israel

Ariel Felner
ISE Department
Ben-Gurion University
Israel

Abstract

Rational lazy A^* is a heuristic search algorithm which combines two or more heuristics efficiently, where one is cheap and less informative, and the other is expensive and more informative. Rational lazy A^* performs meta-reasoning to predict when the added pruning power from the more expensive heuristic is worth the extra time needed to compute it. In this paper, we present the rational lazy A^* planner, which was submitted to the sequential optimal track of the 2014 International Planning Competition.[1]

Introduction

The A^* algorithm (Hart, Nilsson, and Raphael 1968) is a best-first heuristic search algorithm guided by the cost function $f(n) = g(n) + h(n)$. If the heuristic $h(n)$ is admissible (never overestimates the real cost to the goal) then the set of nodes expanded by A^* is both necessary and sufficient to find the optimal path to the goal (Dechter and Pearl 1985). This paper examines the case where we have several available admissible heuristics. Clearly, we can evaluate all these heuristics, and use their *maximum* as an admissible heuristic, a scheme we call A^*_{MAX}. The problem with naive maximization is that all the heuristics are computed for all the generated nodes. In order to reduce the time spent on heuristic computations, Lazy A^* (or LA^*, for short) evaluates the heuristics one at a time, lazily. When a node n is generated, LA^* only computes one heuristic, $h_1(n)$, and adds n to OPEN. Only when n re-emerges as the top of OPEN is another heuristic, $h_2(n)$, evaluated; if this results in an increased heuristic estimate, n is re-inserted into OPEN. This idea was briefly mentioned by Zhang and Bacchus (2012) in the context of the MAXSAT heuristic for planning domains. LA^* is as informative as A^*_{MAX}, but can significantly reduce search time, as we will not need to compute h_2 for many nodes.

LA^* reduces the search time, while maintaining the informativeness of A^*_{MAX}. However, as noted by Domshlak, Karpas, and Markovitch (2012), if the goal is to reduce search time, it may be better to compute a fast heuristic on several nodes, rather than to compute a slow but informative heuristic on only one node. Based on this idea, they formulated *selective max* (Sel-MAX), an online learning scheme which chooses one heuristic to compute at each state. Sel-MAX chooses to compute the more expensive heuristic h_2 for node n when its classifier predicts that $h_2(n) - h_1(n)$ is greater than some threshold, which is a function of heuristic computation times and the average branching factor. Felner et al. (2011) showed that randomizing a heuristic and applying *bidirectional pathmax* (BPMX) might sometimes be faster than evaluating all heuristics and taking the maximum. This technique is only useful in undirected graphs, and is therefore not applicable to some of the domains in this paper. Both Sel-MAX and Random compute the resulting heuristic *once*, before each node is added to OPEN while LA^* computes the heuristic lazily, in different steps of the search. In addition, both randomization and Sel-MAX save heuristic computations and thus reduce search time in many cases. However, they might be less informed than pure maximization and as a result expand a larger number of nodes.

We then combine the ideas of lazy heuristic evaluation and of trading off more node expansions for less heuristic computation time, into a *new* variant of LA^* called *rational lazy* A^* (RLA^*). RLA^* is based on rational meta-reasoning, and uses a myopic *value-of-information* criterion to decide whether to compute $h_2(n)$ or to bypass the computation of h_2 and expand n immediately when n re-emerges from OPEN. RLA^* aims to reduce search time, even at the expense of more node expansions than A^*_{MAX}.

Lazy A^*

Throughout this paper we assume for clarity that we have two available admissible heuristics, h_1 and h_2. Extension to multiple heuristics is straightforward, at least for LA^*. Unless stated otherwise, we assume that h_1 is faster to compute than h_2 but that h_2 is *weakly more informed*, i.e., $h_1(n) \leq h_2(n)$ for the majority of the nodes n, although counter cases where $h_1(n) > h_2(n)$ are possible. We say that h_2 *dominates* h_1, if such counter cases do not exist and $h_2(n) \geq h_1(n)$ for *all* nodes n. We use $f_1(n)$ to denote $g(n) + h_1(n)$. Likewise, $f_2(n)$ denotes $g(n) + h_2(n)$, and $f_{max}(n)$ denotes $g(n) + \max(h_1(n), h_2(n))$. We denote the cost of the optimal solution by C^*. Additionally, we denote the computation time of h_1 and of h_2 by t_1 and t_2, respectively and denote the overhead of an *insert/pop* operation in

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1]This paper is based upon the original publication by Tolpin et al. (2013).

Algorithm 1: Lazy A^*

```
   Input: LAZY-A*
1  Apply all heuristics to Start
2  Insert Start into OPEN
3  while OPEN not empty do
4      n ← best node from OPEN
5      if Goal(n) then
6          return trace(n)
7      if h2 was not applied to n then
8          Apply h2 to n
9          insert n into OPEN
10         continue       //next node in OPEN
11     foreach child c of n do
12         Apply h1 to c.
13         insert c into OPEN
14     Insert n into CLOSED
15 return FAILURE
```

OPEN by t_o. Unless stated otherwise we assume that t_2 is much greater than $t_1 + t_o$. LA^* thus mainly aims to reduce computations of h_2.

The pseudo-code for LA^* is depicted as Algorithm 1, and is very similar to A^*. In fact, without lines 7 – 10, LA^* would be identical to A^* using the h_1 heuristic. When a node n is generated we only compute $h_1(n)$ and n is added to OPEN (Lines 11 – 13), without computing $h_2(n)$ yet. When n is first removed from OPEN (Lines 7 – 10), we compute $h_2(n)$ and reinsert it into OPEN, this time with $f_{max}(n)$.

It is easy to see that LA^* is as informative as A^*_{MAX}, in the sense that both A^*_{MAX} and LA^*expand a node n only if $f_{max}(n)$ is the best f-value in OPEN. Therefore, LA^* and A^*_{MAX} generate and expand and the same set of nodes, up to differences caused by tie-breaking.

In its general form A^* generates many nodes that it does not expand. These nodes, called *surplus* nodes (Felner et al. 2012), are in OPEN when we expand the goal node with $f = C^*$. All nodes in OPEN with $f > C^*$ are surely surplus but some nodes with $f = C^*$ may also be surplus. The number of surplus nodes in OPEN can grow exponentially in the size of the domain, resulting in significant costs.

LA^* avoids h_2 computations for many of these surplus nodes. Consider a node n that is generated with $f_1(n) > C^*$. This node is inserted into OPEN but will never reach the top of OPEN, as the goal node will be found with $f = C^*$. In fact, if OPEN breaks ties in favor of small h-values, the goal node with $f = C^*$ will be expanded as soon as it is generated and such savings of h_2 will be obtained for some nodes with $f_1 = C^*$ too. We refer to such nodes where we saved the computation of h_2 as *good* nodes. Other nodes, those with $f_1(n) < C^*$ (and some with $f_1(n) = C^*$) are called *regular nodes* as we apply both heuristics to them.

A^*_{MAX} computes both h_1 and h_2 for all generated nodes, spending time t_1+t_2 on all generated nodes. By contrast, for *good* nodes LA^* only spends t_1, and saves t_2. In the basic implementation of LA^* (as in algorithm 1) *regular* nodes are inserted into OPEN twice, first for h_1 (Line 13) and then for h_2 (Line 9) while *good* nodes only enter OPEN once (Line 13). Thus, LA^* has some extra overhead of OPEN operations for *regular nodes*. We distinguish between 3 classes of nodes:

(1) *expanded regular* (ER) — nodes that were expanded after both heuristics were computed.

(2) *surplus regular* (SR) — nodes for which h_2 was computed but are still in OPEN when the goal was found.

(3) *surplus good* (SG) — nodes for which only h_1 was computed by LA^* when the goal was found.

Alg	ER	SR	SG
A^*_{MAX}	$t_1 + \mathbf{t_2} + 2t_o$	$t_1 + \mathbf{t_2} + t_o$	$t_1 + \mathbf{t_2} + t_o$
LA^*	$t_1 + \mathbf{t_2} + 4t_o$	$t_1 + \mathbf{t_2} + 3t_o$	$t_1 + t_o$

Table 1: Time overhead for A^*_{MAX} and for LA^*

The time overhead of A^*_{MAX} and LA^* is summarized in Table 1. LA^* incurs more OPEN operations overhead, but saves h_2 computations for the SG nodes. When t_2 (**boldface** in table 1) is significantly greater than both t_1 and t_o there is a clear advantage for LA^*, as seen in the SG column.

Rational Lazy A^*

LA^* offers us a very strong guarantee, of expanding the same set of nodes as A^*_{MAX}. However, often we would prefer to expand more states, if it means reducing search time. We now present *Rational Lazy A** (RLA^*), an algorithm which attempts to optimally manage this tradeoff.

Using principles of rational meta-reasoning (Russell and Wefald 1991), theoretically every algorithm action (heuristic function evaluation, node expansion, open list operation) should be treated as an action in a sequential decision-making meta-level problem: actions should be chosen so as to achieve the minimal expected search time. However, the appropriate general meta-reasoning problem is extremely hard to define precisely and to solve optimally.

Therefore, we focus on just one decision type, made in the context of LA^*, when n re-emerges from OPEN (Line 7). We have two options: **(1)** Evaluate the second heuristic $h_2(n)$ and add the node back to OPEN (Lines 7-10) like LA^*, or **(2)** bypass the computation of $h_2(n)$ and expand n right way (Lines 11 -13), thereby saving time by not computing h_2, at the risk of additional expansions and evaluations of h_1. In order to choose rationally, we define a criterion based on value of information (VOI) of evaluating $h_2(n)$ in this context.

The only addition of RLA^* to LA^* is the option to bypass h_2 computations (Lines 7-10). Suppose that we choose to compute h_2 — this results in one of the following outcomes:

1: n is still expanded, either now or eventually.

2: n is re-inserted into OPEN, and the goal is found without ever expanding n.

Computing h_2 is helpful only in outcome 2, where potential time savings are due to pruning a search subtree at the expense of the time $t_2(n)$. However, whether outcome 2 takes place after a given state is not known to the algorithm until the goal is found, and the algorithm must decide whether to evaluate h_2 according to what it *believes to be* the probability of each of the outcomes. We derive a *rational policy* for when to evaluate h_2, under the myopic assumption

that the algorithm continues to behave like LA^* afterwards (i.e., it will never again consider bypassing the computation of h_2).

The time wasted by being sub-optimal in deciding whether to evaluate h_2 is called the *regret* of the decision. If $h_2(n)$ is not helpful and we decide to compute it, the effort for evaluating $h_2(n)$ turns out to be wasted. On the other hand, if $h_2(n)$ is helpful but we decide to bypass it, we needlessly expand n. Due to the myopic assumption, RLA^* would evaluate both h_1 and h_2 for all successors of n.

	Compute h_2	Bypass h_2
h_2 helpful	0	$t_e + (b(n) - 1)t_d$
h_2 not helpful	t_d	0

Table 2: Regret in Rational Lazy A*

Table 2 summarizes the regret of each possible decision, for each possible future outcome; each column in the table represents a decision, while each row represents a future outcome. In the table, t_d is the to time compute h_2 and re-insert n into OPEN thus delaying the expansion of n, t_e is the time to remove n from OPEN, expand n, evaluate h_1 on each of the $b(n)$ ("local branching factor") children $\{n'\}$ of n, and insert $\{n'\}$ into the open list. Computing h_2 needlessly wastes time t_d. Bypassing h_2 computation when h_2 would have been helpful wastes $t_e + b(n)t_d$ time, but because computing h_2 would have cost t_d, the regret is $t_e + (b(n) - 1)t_d$.

Let us denote the probability that h_2 is helpful by p_h. The expected regret of computing h_2 is thus $(1 - p_h)t_d$. On the other hand, the expected regret of bypassing h_2 is $p_h(t_e + (b(n) - 1)t_d)$. As we wish to minimize the expected regret, we should thus evaluate h_2 just when:

$$(1 - p_h)t_d < p_h(t_e + (b(n) - 1)t_d) \tag{1}$$

or equivalently:

$$(1 - b(n)p_h)t_d < p_h t_e \tag{2}$$

If $p_h b(n) \geq 1$, then the expected regret is minimized by always evaluating h_2, regardless of the values of t_d and t_e. In these cases, RLA^* cannot be expected to do better than LA^*. For example, in the 15-puzzle and its variants, the effective branching factor is ≈ 2. Therefore, if h_2 is expected to be helpful for more than half of the nodes n on which LA^* evaluates $h_2(n)$, then one should simply use LA^*.

For $p_h b(n) < 1$, the decision of whether to evaluate h_2 depends on the values of t_d and t_e:

$$\textbf{evaluate } h_2 \textbf{ if } t_d < \frac{p_h}{1 - p_h b(n)} t_e \tag{3}$$

Denote by t_c the time to generate the children of n. Then:

$$\begin{aligned} t_d &= t_2 + t_o \\ t_e &= t_o + t_c + b(n)t_1 + b(n)t_o \end{aligned} \tag{4}$$

By substituting (4) into (3), obtain: **evaluate** h_2 **if**:

$$t_2 + t_o < \frac{p_h \left[t_c + b(n)t_1 + (b(n) + 1)t_o\right]}{1 - p_h b(n)} \tag{5}$$

The factor $\frac{p_h}{1 - p_h b(n)}$ depends on the potentially unknown probability p_h, making it difficult to reach the optimum decision. However, if our goal is just to do better than LA^*, then it is safe to replace p_h by an upper bound on p_h. Note that the values p_h, t_1, t_2, t_c may actually be variables that depend in complicated ways on the state of the search. Despite that, the very crude model we use, assuming that they are setting-specific constants, is sufficient to achieve improved performance.

We now turn to implementation-specific estimation of the runtimes. OPEN in A^* is frequently implemented as a priority queue, and thus we have, approximately, $t_o = \tau \log N_o$ for some τ, where the size of OPEN is N_o. Evaluating h_1 is cheap in many domains, as is the case with Manhattan Distance (MD) in discrete domains, t_o is the most significant part of t_e. In such cases, rule (5) can be approximated as 6:

$$\textbf{evaluate } h_2 \textbf{ if } t_2 < \frac{\tau p_h}{1 - p_h b(n)} (b(n) + 1) \log N_o \tag{6}$$

Rule (6) recommends to evaluate h_2 mostly at late stages of the search, when the open list is large, and in nodes with a higher branching factor.

In other domains, such as planning, both t_1 and t_2 are significantly greater than both t_o and t_c, and terms not involving t_1 or t_2 can be dropped from (5), resulting in Rule (7):

$$\textbf{evaluate } h_2 \textbf{ if } \frac{t_2}{t_1} < \frac{p_h b(n)}{1 - p_h b(n)} \tag{7}$$

The right hand side of (7) grows with $b(n)$, and here it is beneficial to evaluate h_2 only for nodes with a sufficiently large branching factor.

IPC 2014 Submission

We implemented LA^* and RLA^* on top of the Fast Downward planning system (Helmert 2006). Ideally, the two heuristics we use are: the admissible landmarks heuristic h_{LA} (used as h_1) (Karpas and Domshlak 2009), and the landmark cut heuristic h_{LMCUT} (Helmert and Domshlak 2009) (used as h_2).

However, IPC 2014 introduced conditional effects as a required feature to support. We have developed a variant of h_{LMCUT} which support conditional effects, by relaxing each operator into several unary effect operators — one for each effect, which also includes the condition for that effect. However, supporting conditional effects in h_{LA} requires a great deal of work. Therefore, when a planning task with conditional effects is given to our planner, it uses h_{max} (Bonet, Loerincs, and Geffner 1997) as h_1, and the modified version of h_{LMCUT} as h_2. For planning tasks with no conditional effects, our planner still uses h_{LA} and the original version of h_{LMCUT}.

When applying RLA^* in planning domains we evaluate rule (7) at every state. This rule involves two unknown quantities: $\frac{t_2}{t_1}$, the ratio between heuristic computations times, and p_h, the probability that h_2 is helpful. Estimating $\frac{t_2}{t_1}$ is quite easy — we simply use the average computation times of both heuristics, which we measure as search progresses.

Estimating p_h is not as simple. While it is possible to empirically determine the best value for p_h, this does not

fit the paradigm of domain-independent planning. Furthermore, planning domains are very different from each other, and even problem instances in the same domain are of varying size, and thus getting a single value for p_h which works well for many problems is difficult. Instead, we vary our estimate of p_h adaptively during search. To understand this estimate, first note that if n is a node at which h_2 was helpful, then we computed h_2 for n, but did not expand n. Thus, we can use the number of states for which we computed h_2 that were not yet expanded (denoted by A), divided by the number of states for which we computed h_2 (denoted by B), as an approximation of p_h. However, $\frac{A}{B}$ is not likely to be a stable estimate at the beginning of the search, as A and B are both small numbers. To overcome this problem, we "imagine" we have observed k examples, which give us an estimate of $p_h = p_{init}$, and use a weighted average between these k examples, and the observed examples — that is, we estimate p_h by $(\frac{A}{B} \cdot B + p_{init} \cdot k)/(B + k)$. In our empirical evaluation, we used $k = 1000$ and $p_{init} = 0.5$.

Conclusion

We discussed two schemes for decreasing heuristic evaluation times. LA^* is very simple to implement and is as informative as A^*_{MAX}. LA^* can significantly speed up the search, especially if t_2 dominates the other time costs. Rational LA^* allows additional cuts in h_2 evaluations, at the expense of being less informed than A^*_{MAX}. However, due to a rational tradeoff, this allows for an additional speedup, and in our empirical evaluation (Tolpin et al. 2013) Rational LA^* achieves the best overall performance in our domains.

RLA^* is simpler to implement than its direct competitor, Sel-MAX, but its decision can be more informed. When RLA^* has to decide whether to compute h_2 for some node n, it already *knows* that $f_1(n) \leq C^*$. By contrast, although Sel-MAX uses a much more complicated decision rule, it makes its decision when n is first generated, and does not know whether h_1 will be informative enough to prune n. Rational LA^* outperforms Sel-MAX in our planning experiments.

RLA^* and its analysis can be seen as an instance of the rational meta-reasoning framework (Russell and Wefald 1991). While this framework is very general, it is extremely hard to apply in practice. Recent work exists on meta-reasoning in DFS algorithms for CSP) (Tolpin and Shimony 2011) and in Monte-Carlo tree search (Hay et al. 2012). This paper applies these methods successfully to a variant of A^*. There are numerous other ways to use rational meta-reasoning to improve A^*, starting from generalizing RLA^* to handle more than two heuristics, to using the meta-level to control decisions in other variants of A^*. All these potential extensions provide fruitful ground for future work.

References

Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A robust and fast action selection mechanism for planning. In Kuipers, B., and Webber, B. L., eds., *AAAI/IAAI*, 714–719. AAAI Press / The MIT Press.

Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM* 32(3):505–536.

Domshlak, C.; Karpas, E.; and Markovitch, S. 2012. Online speedup learning for optimal planning. *JAIR* 44:709–755.

Felner, A.; Zahavi, U.; Holte, R.; Schaeffer, J.; Sturtevant, N.; and Zhang, Z. 2011. Inconsistent heuristics in theory and practice. *Artificial Intelligence* 175(9-10):1570–1603.

Felner, A.; Goldenberg, M.; Sharon, G.; Stern, R.; Beja, T.; Sturtevant, N. R.; Schaeffer, J.; and R, H. 2012. Partial-expansion a* with selective node generation. In *AAAI*, 471–477.

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* SCC-4(2):100–107.

Hay, N.; Russell, S.; Tolpin, D.; and Shimony, S. E. 2012. Selecting computations: Theory and applications. In de Freitas, N., and Murphy, K. P., eds., *UAI*, 346–355. AUAI Press.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*, 162–169.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In *IJCAI*, 1728–1733.

Russell, S., and Wefald, E. 1991. Principles of metereasoning. *Artificial Intelligence* 49:361–395.

Tolpin, D., and Shimony, S. E. 2011. Rational deployment of CSP heuristics. In Walsh, T., ed., *IJCAI*, 680–686. IJCAI/AAAI.

Tolpin, D.; Beja, T.; Shimony, S. E.; Felner, A.; and Karpas, E. 2013. Toward rational deployment of multiple heuristics in a. In Rossi, F., ed., *IJCAI*. IJCAI/AAAI.

Zhang, L., and Bacchus, F. 2012. Maxsat heuristics for cost optimal planning. In *AAAI*.

SPM&S Planner: Symbolic Perimeter Merge-and-Shrink

Álvaro Torralba and **Vidal Alcázar** and **Carlos Linares López** and **Daniel Borrajo**
{alvaro.torralba, vidal.alcazar, carlos.linares, daniel.borrajo}@uc3m.es
Universidad Carlos III de Madrid
Madrid, Spain

Peter Kissmann
kissmann@cs.uni-saarland.de
Saarland University
Saarbrücken, Germany

Stefan Edelkamp
edelkamp@tzi.de
University of Bremen
Bremen, Germany

Abstract

The SPM&S planner uses Symbolic Perimeter Merge-and-Shrink and Symbolic Perimeter Pattern Database heuristics within a standard A* search. This paper briefly describes the heuristics used and some technical details of the planner.

Introduction

One of the most successful approaches to cost-optimal planning is the use of abstraction heuristics as admissible estimators in an A* search. Abstraction heuristics use the optimal cost of the solution in a simplified abstract state space as an admissible estimation of the cost of the original problem. The heuristic value of each abstract state is usually precomputed and stored in a lookup table. We consider two automatic ways to derive domain-independent abstractions: Pattern Databases (Culberson and Schaeffer 1998; Edelkamp 2001) and Merge-and-Shrink (Helmert, Haslum, and Hoffmann 2007; Helmert et al. 2014). Pattern Databases (PDBs) perform a projection of the planning task over a subset of variables. Merge-and-Shrink (M&S) is a flexible approach that allows deriving abstractions more general than PDBs.

Perimeter search (Dillenburg and Nelson 1994; Manzini 1995) is a form of bidirectional search that operates in two phases: the backward phase and the forward phase. The backward phase generates a perimeter around the goal with a uniform-cost (Dijkstra) search. Then, the forward phase performs forward search from the initial state to the perimeter. Since the goal of the forward search is any state in the perimeter P, the heuristic estimates the distance to the closest state in the perimeter so that $h(s) = \min_{s' \in P} h(s, s')$. Perimeter PDBs (Felner and Ofek 2007) estimate the distance to the closest node in the perimeter. They were initially proposed in the context of heuristic search for combinatorial puzzles (Felner and Ofek 2007) and later adapted to automated planning in a work parallel with ours (Eyerich and Helmert 2013).

Symbolic PDBs (Edelkamp 2002) are different from regular PDBs in that they are stored as Binary Decision Diagrams (BDDs) (Bryant 1986) and that symbolic search is used to traverse the abstract space. The GAMER planner uses symbolic A* with Symbolic PDBs (Kissmann and Edelkamp 2011). Previous work on Symbolic PDBs have shown their ability to derive stronger heuristics (Kissmann 2012) by exploring larger, less-abstracted, state spaces, able to compete with M&S heuristics (Edelkamp, Kissmann, and Torralba 2012). More recently, optimizations on image computation (Torralba, Edelkamp, and Kissmann 2013) and the use of constraints from state invariants to prune the search (Torralba and Alcázar 2013) have further improved the performance in symbolic search, which directly affects the accuracy of symbolic heuristics.

The Symbolic Perimeter M&S planner (SPM&S) implements a new algorithm that combines abstraction heuristics, perimeter search and symbolic search to derive admissible estimates. The algorithm was originally presented in (Torralba, Linares López, and Borrajo 2013) [1]. In this paper, we only explain our approach briefly and the parameter configuration used for IPC-2014.

Symbolic Perimeter Merge-and-Shrink

Our proposed technique, SPM&S, derives a symbolic heuristic using M&S abstractions to relax a symbolic backward search. Figure 1 represents a high level view of the interaction between symbolic search and M&S abstractions in the SPM&S algorithm. SPM&S starts computing a symbolic perimeter, $Exp(\alpha_0)$. The first BDD with $h = 0$ contains the goal states and by successive *pre-image* operations SPM&S generates the sets of states with $h = 1, h = 2$, etc. Exploring the whole search space is not practical for general planning domains, so when memory or time bounds are surpassed, it is truncated ($h = 2$ in Figure 1). The minimum distance to the goal of the expanded state sets is stored, transforming the list of BDDs representing the search into an Algebraic Decision Diagram (ADD) (Bahar et al. 1997) representing the heuristic, $h_{Exp(\alpha_0)}$.

Then, M&S is used to derive an abstraction, α_1. M&S merges variables, applying shrinking if needed to fit the maximum number of abstract states (in Figure 1, α_1 must have at most three abstract states before merging the next variable). SPM&S uses α_1 to relax the top levels of all the BDDs in the exploration $Exp(\alpha_1)$. All partial states related to the same abstract state are considered equivalent so that,

[1] We originally called our technique Symbolic M&S in (2013). We rename it to Symbolic Perimeter M&S to highlight its condition of perimeter abstraction heuristic.

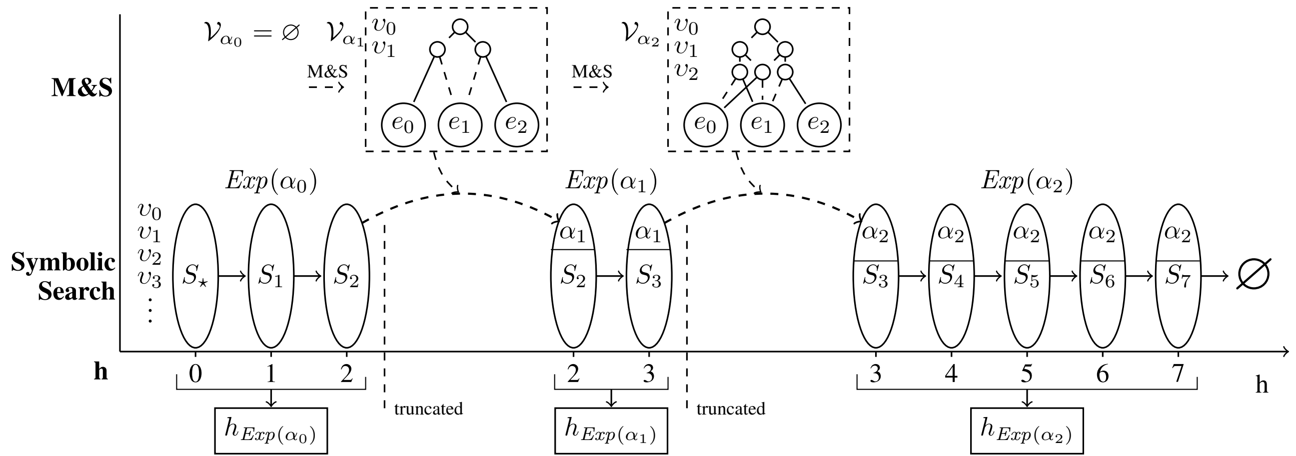


Figure 1: SM&S example with binary variables, unary cost operators and a limit of six abstract states for M&S.

when one is reached, all of them are. In Figure 1, abstract state e_1 represents partial states 00 and 10. During the exploration, if state 10010… is reached, then state 00010… is also reached and *vice versa*. Hence, BDD nodes pointed to by 00 and 10 are equivalent, making the top part of any BDD in the exploration equal to its M&S representation. Also, M&S abstractions are accumulative, so the top levels of α_2 coincide with those of previous abstractions. SPM&S continues interleaving symbolic explorations and M&S iterations until an exploration is completed or time/memory bounds are violated. When finished, it returns the list of ADDs representing the heuristic.

Memory is controlled by the maximum number of M&S abstract states N and the maximum number of nodes N_F to represent the search frontier. Two different parameters limit time: $T_{\text{M\&S}}$ is the total time allowed for the generation of the heuristic, and T_I limits the maximum time employed in one $pre\text{-}image$ operation. If T_I is exceeded, not only the $pre\text{-}image$ but the whole exploration is halted. In order to avoid starting another $image$ as hard as the halted one, the maximum number of nodes in the frontier search is reduced to half of the size of the current frontier that we just failed to expand.

PDB selection

The SPM&S algorithm can also be used in combination with Pattern Database abstractions, as they are a particular case of M&S abstraction. For the pattern selection, instead of choosing a particular fixed pattern for the abstraction like previous domain-independent methods (Haslum et al. 2007; Kissmann and Edelkamp 2011; Edelkamp 2006), we seek a complete hierarchy. This hierarchy consists of an ordering on the variables such that the variables at the beginning are abstracted first until a small enough abstraction is obtained. This ordering is similar to the one used by linear merge strategies of M&S, although in this case it is reversed, as the merge strategies consider important variables first whereas we abstract away important variables last.

These orderings are computed by adding goal variables to the ordering (Goal first strategies) or variables causally connected to variables already in the ordering (CG first strategies). Tie-breaking is performed either randomly (Random) or by taking Gamer's ordering (Gamer) or its reverse (Reverse Gamer). The SPM&S planner makes use of three different strategies, derived from the previously described criteria:

- CG Goal Random: Adds variables causally connected to variables already in the ordering. If there is none, a goal variable is added. Ties are broken randomly.
- Goal CG Gamer: Goal variables are added first. Then, variables causally connected to variables already in the ordering are added. Ties are broken by adding to the ordering variables lower in Gamer's ordering.
- Reverse Gamer: The ordering is the same as Gamer's, but reversed. This aims to obtain abstractions that are efficiently computable, as the variables are abstracted away from the top of the BDD. Goals and causal links are explicitly ignored, although they are taken into account in Gamer's ordering. Also, at some point the abstraction may not contain goal variables, although this is fine as long as we work with a perimeter.

The SPM&S Planner

The SPM&S planner is implemented on top of the Fast Downward planning system (Helmert 2006). The SPM&S planner generates several heuristics during the precomputation phase. Then, it performs an explicit-state A* search taking the maximum of all the precomputed heuristics. The SPM&S planner precomputes up to five abstractions, although the perimeter in the original state space is computed only once and then reused for subsequent applications of the algorithm. Each run of the algorithm uses different abstraction procedures until the available time has been exhausted or the five strategies have been used. The strategies are used in the following order:

1. M&S using bisimulation shrinking, with a maximum number of 10000 abstract states
2. PDBs with CG Goal Random

3. PDBs with Goal CG Gamer
4. PDBs with Reverse Gamer
5. PDBs with CG Goal Random

Note that the second and fifth strategies are the same, but they typically produce different results due to the random tie-breaking.

Preprocessing

There are two noteworthy considerations regarding the preprocessing phase in SPM&S:

- How the SAS$^+$ variables are selected.
- How to compute h^2 (Bonet and Geffner 2001) and prune spurious operators.

SAS$^+$ Variable Selection

Switching from Gamer's SAS$^+$ encoding to the Fast Downward version (Helmert 2009), we observed a decrease of performance in some benchmark domains. We changed the selection of which invariant groups are used as SAS$^+$ variables in order to avoid that degradation in performance.

The Fast Downward planner chooses invariant groups with the highest cardinality as SAS$^+$ variables, until all the fluents of the problem have been considered in a variable. Aiming to further reduce the number of SAS$^+$ variables selected, we prefer to select invariant groups that contain fluents that do not appear in other invariant groups. We base our criterion on the observation that, since all the fluents of the problem have to be included in a SAS$^+$ variable, invariant groups that have a fluent which does not appear in other invariant groups will always be selected anyway.

As an example, think of the following case, based on a simplified version of the IPC-2011 *floortile* domain: we have two robots on a grid such that the robots cannot be at the same cell at the same time. Two types of invariant groups are detected:

1. Each robot is at exactly one single cell:
   (at robot1 cell1),(at robot1 cell2),...
2. Each cell either is clear or has a robot at it:
   (clear cell1),(at robot1 cell1),(at robot2 cell1)

Invariants of the first type have larger cardinality, so Fast Downward would encode this problem with a variable per robot that represents the location of the robot ({*(at robot1 cell1),(at robot1 cell2),...*}) and a variable of the kind {*(clear cell1),⟨none of those⟩*} per cell. In our case, we prefer to select invariant groups of the second type first because each fluent *(clear cell1)* only takes part on a single invariant group. Thus, we would only have a variable per cell of the kind {(clear cell1),(at robot1 cell1),(at robot2 cell1)}, which amounts to fewer variables and fluents.

This leads to the use of *"exactly-one"* invariant groups as variables in most cases, avoiding the use of *"at-most-one"* invariant groups if possible – which require an additional ⟨*none of those*⟩ fluent. With this policy the number of resulting variables and fluents is usually lower. This may be counterproductive for techniques that depend on the causal graph, like the abstraction strategies that we use.

Computing h^2 Invariants and Pruning Spurious Operators

We have implemented the computation of the h^2 in Fast Downward's preprocessor. We also implemented a backward version of h^2 (Haslum 2008), which identifies pairs of propositions that cannot be reached from goal states in regression.

We use the mutexes obtained from h^2 and the *"exactly-one"* invariant groups from Fast Downward's monotonicity analysis to disambiguate the preconditions and the effects of the operators of the problem (Alcázar et al. 2013). We discard operators whose preconditions or effects are spurious sets of fluents, that is, contradict the previously inferred state invariants. We do this because the number of ground operators is significantly reduced in many planning domains with respect to the standard preprocessor of Fast Downward.

The discovery and use of the state invariants during this phase is interleaved: whenever new mutexes or spurious operators are discovered in this process, we repeat the computation of h^2 in both directions and the operator disambiguation until no more constraints are inferred. We set a limit of 300 seconds for this phase.

Parameter Configuration

SPM&S uses a total of $T_{SM\&S} = 1200$ seconds and 3GB for the precomputation phase. When any of those bounds are violated the precomputation phase automatically ends and the planner starts an A* search with the heuristics that were successfully precomputed so far.

During the precomputation phase, SPM&S perform several regression searches in different state spaces. Each search continues until the frontier has more than $N_F = 10$ million nodes or the next step is estimated to take $T_I = 30$ seconds. Whenever a search surpasses any of these bounds, it is deemed as "not searchable" and its used as a perimeter to initialize a new search on a more abstracted state space.

All symbolic searches performed in the precomputation phase use the latest improvements on image computation (Torralba, Edelkamp, and Kissmann 2013) and h^2 constraints for symbolic search (Torralba and Alcázar 2013). In particular, we use a disjunctive partitioning of TRs with a maximum TR size of 100k nodes and a timeout of 60 seconds to generate the TRs. Constraints from the state invariants are encoded directly in the TRs.

Conditional Effect Support

Fast Downward planning system partially supports conditional effects. They are correctly handled by the A* search, but the current public version of M&S does not support them. We chose to disable M&S relaxations on these cases, so only Symbolic Perimeter PDBs were used on domains with conditional effects.

To handle conditional effects in symbolic search, we encode them in the TRs. According with the semantics of conditional effects, they are applied in order. If more than one effect is applied over the same variable, the last one overwrites all the others. Hence, to generate the TR with conditional effects we group all the effects by the variable they

modify. Each conditional effect is encoded as the conjunction of its condition, its effect and the negation of the conditions of previous effects over the same variable.

For the h^2 computation, conditional effects are not compiled away, but rather we ignore conditional preconditions and all the delete effects that are conditional or that delete a conditional effect.

Technical Details

To perform BDD operations, we used version 2.5 of Fabio Somenzi's CUDD library. The planner is compiled in 32-bit (-m32), using the compiler optimization (-O3) and with support of c++-11 features (–std=c++11).

Acknowledgements

This work has been partially supported by a FPI grant from the Spanish government associated to the MICINN project TIN2008-06701-C03-03, and it has also been supported by the project TIN2011-27652-C03-02.

References

Alcázar, V.; Borrajo, D.; Fernández, S.; and Fuentetaja, R. 2013. Revisiting regression in planning. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Bahar, R. I.; Frohm, E. A.; Gaona, C. M.; Hachtel, G. D.; Macii, E.; Pardo, A.; and Somenzi, F. 1997. Algebraic decision diagrams and their applications. *Formal Methods in System Design* 10(2/3):171–206.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

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.

Dillenburg, J. F., and Nelson, P. C. 1994. Perimeter search. *Artificial Intelligence* 65(1):165–178.

Edelkamp, S.; Kissmann, P.; and Torralba, Á. 2012. Symbolic A* search with pattern databases and the merge-and-shrink abstraction. In *European Conference on Artificial Intelligence (ECAI)*, 306–311.

Edelkamp, S. 2001. Planning with pattern databases. In *ECP*, 13–34.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *Conference on Artificial Intelligence Planning Systems (AIPS)*, 274–283.

Edelkamp, S. 2006. Automated creation of pattern database search heuristics. In *MoChArt*, 35–50.

Eyerich, P., and Helmert, M. 2013. Stronger abstraction heuristics through perimeter search. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

Felner, A., and Ofek, N. 2007. Combining perimeter search and pattern database abstractions. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, 155–168.

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 Conference on Artificial Intelligence (AAAI)*, 1007–1012.

Haslum, P. 2008. Additive and reversed relaxed reachability heuristics revisited. In *International Planning Competition (IPC)*.

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 Association for Computing Machinery*. Accepted.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 176–183.

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

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5-6):503–535.

Kissmann, P., and Edelkamp, S. 2011. Improving cost-optimal domain-independent symbolic planning. In *AAAI Conference on Artificial Intelligence (AAAI)*.

Kissmann, P. 2012. *Symbolic Search in Planning and General Game Playing*. Ph.D. Dissertation, Universität Bremen.

Manzini, G. 1995. Bida: An improved perimeter search algorithm. *Artificial Intelligence* 75(2):347–360.

Torralba, Á., and Alcázar, V. 2013. Constrained symbolic search: On mutexes, bdd minimization and more. In *Symposium on Combinatorial Search (SoCS)*.

Torralba, Á.; Edelkamp, S.; and Kissmann, P. 2013. Transition trees for cost-optimal symbolic planning. In *International Conference on Automated Planning and Scheduling (ICAPS)*.

Torralba, Á.; Linares López, C.; and Borrajo, D. 2013. Symbolic merge-and-shrink for cost-optimal planning. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

SymBA*: A Symbolic Bidirectional A* Planner

Álvaro Torralba and **Vidal Alcázar** and **Daniel Borrajo**
{alvaro.torralba, vidal.alcazar, daniel.borrajo}@uc3m.es
Universidad Carlos III de Madrid
Madrid, Spain

Peter Kissmann
kissmann@cs.uni-saarland.de
Saarland University
Saarbrücken, Germany

Stefan Edelkamp
edelkamp@tzi.de
University of Bremen
Bremen, Germany

Abstract

Lately, several important advancements have been obtained in symbolic search. First, bidirectional blind search has obtained good results on many domains. Second, perimeter-based abstraction heuristics have been proposed as an important improvement over regular abstraction heuristics. Motivated by the synergy between bidirectional search and perimeter-based abstraction heuristics, here we present SymBA*, which performs bidirectional A* using the frontiers of the opposite search to infer informed perimeter-based abstraction heuristics.

Motivation

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 collision between frontiers (Alcázar, Fernández, and Borrajo 2014). However, symbolic search (McMillan 1993) reasons using sets of states, avoiding the otherwise complex problem of subsumption of states and substantially reducing the cost of detecting the collision of frontiers. Moreover, recent advances in symbolic search planning have helped to overcome some of the problems of performing regression in planning (Torralba and Alcázar 2013). Thus, recent results have shown that symbolic bidirectional blind search has become one of the best alternatives for cost-optimal planning, outperforming not only A*-based planners but also BDDA*, the symbolic search variant of A*.

These recent improvements have risen the question of whether it is possible to use heuristics in combination with symbolic bidirectional search. Bidirectional heuristic search has a long history (Pohl 1969; de Champeaux 1983), but the hardness of proving optimality reduces considerably the advantages it has over regular A* (Kaindl and Kainz 1997). Because of this, bidirectional heuristic search has fallen out of flavor, with the majority of the search and planning community working mostly with regular A*.

Abstraction heuristics, like Pattern Databases (PDBs) (Culberson and Schaeffer 1998) and Merge-and-Shrink (M&S) (Helmert et al. 2014), are commonly used admissible heuristics. These heuristics can be enhanced with a perimeter (Felner and Ofek 2007; Eyerich and Helmert 2013; Torralba, Linares López, and Borrajo 2013), which leads to a strictly more informed heuristic than both the abstraction heuristic and the perimeter alone. Now, abstraction heuristics do not require a perimeter of a fixed radius to obtain better estimates – any frontier in the original space can be used as a seed to improve the heuristic as long as the g value of the expanded states is optimal. Because of this, we propose the use of the frontier in one direction in a bidirectional search algorithm to enhance an abstraction heuristic used by the search in the opposite direction.

The aim of the SymBA* planner is to exploit the synergy between bidirectional search and perimeter-based abstraction heuristics. SymBA* performs bidirectional searches on different state spaces. It starts in the original search space and, when the search becomes too hard, it derives an abstraction heuristic enhanced by the frontier of the opposite direction. The planner decides at any point whether to advance the search in the original state space, enlarging the perimeter, or search in an abstract state space to provide better heuristic estimations for the unabstracted search.[1]

This paper describes the main algorithm used by the planner and technical and implementation details. A more elaborate theoretical discussion will be presented in a future paper.

Symbolic Bidirectional A*

SymBA* performs several symbolic bidirectional A* searches on different state spaces. We denote a bidirectional search on a state space, Θ_i, as S^{Θ_i}. A bidirectional search is composed of two unidirectional searches in opposite directions: a forward search, $S^{\Theta_i}_{fw}$, and a backward search, $S^{\Theta_i}_{bw}$. We will use $S^{\Theta_i}_{u}$ to denote a unidirectional search in an unspecified direction. $f(S^{\Theta_i}_{u})$ denotes the value of the f-layer with minimum f.

First, SymBA* starts a bidirectional search in the original state space. Since no abstraction heuristic has been derived yet, it behaves like symbolic bidirectional blind search.

[1] A connection can be made with hierarchical heuristic search (Holte, Grajkowski, and Tanner 2005), in particular with Switchback (Larsen et al. 2010) and its improved version Short-Circuit (Leighton, Ruml, and Holte 2011), as both traverse the abstract state lazily to avoid searching parts that are irrelevant for the problem at hand.

This search continues until the next step in both directions is deemed as not searchable, because SymBA* estimates that it will take too much time or memory. Only then, a new bidirectional search is initialized in an abstract state space. Both the forward and backward searches are initialized with the frontiers of the current original search. The abstract searches provide heuristic estimations for the original search, increasing the f-value of states in the search frontier. Eventually, the search in the original state space will be simplified (as the number of states with minimum f-value will be smaller)[2]and SymBA* will continue expanding states in the original search space.

A distinction must be made between bidirectional search in the original state space and the ones performed in abstract state spaces. The first type of search aims to find a plan, so techniques like nipping and pruning (Kwa 1989) should remain activated to prune both search frontiers whenever they meet. On the other hand, searches on abstract state spaces are used to derive heuristic estimates for the original search. The additional pruning is deactivated in order to guarantee that the derived heuristics are admissible.

Algorithm 1: SymBA*

1 $S^{\Theta}_{fw} \leftarrow \langle \mathcal{I}, \Theta \rangle$
2 $S^{\Theta}_{bw} \leftarrow \langle \mathcal{G}, \Theta \rangle$
3 $SearchPool \leftarrow \{S^{\Theta}_{fw}, S^{\Theta}_{bw}\}$
4 $\pi \leftarrow None$
5 **while** $\max(f(S^{\Theta}_{fw}), f(S^{\Theta}_{bw})) < Cost(\pi)$ **do**
6   **if** $\exists S \in SearchPool \mid$ `IsCandidate(S)` **then**
7     $S^{\Theta_i}_{u} \leftarrow$ `SelectSearch(`$SearchPool$`)`
8     $\pi' \leftarrow$ `Expand-frontier(`$S^{\Theta_i}_{u}$`)`
9   **if** $\Theta_i = \Theta \wedge \pi' \neq \varnothing \wedge Cost(\pi') < Cost(\pi)$ **then**
10       $\pi \leftarrow \pi'$
11   `Notify-h(`S^{Θ_i}, S^{Θ}`)`
12   **else**
13     $\alpha \leftarrow$`Select-abstraction(`$S^{\Theta}_{fw}, S^{\Theta}_{bw}$`)`
14     $\left\langle S^{\Theta_\alpha}_{fw}, S^{\Theta_\alpha}_{bw} \right\rangle \leftarrow$`Apply(`$\alpha, S^{\Theta}_{fw}, S^{\Theta}_{bw}$`)`
15     $SearchPool \leftarrow SearchPool \cup \{S^{\Theta_\alpha}_{fw}, S^{\Theta_\alpha}_{bw}\}$
16   **return** π

Algorithm 1 shows the main algorithm of SymBA*, which decides whether to advance the search in the original state space or in one of the abstract state spaces. SymBA* maintains a pool of all the current active searches. The pool is initialized with a bidirectional search on the original state space. The algorithm proceeds while the current best solution so far has not been proven optimal (line 5). At each iteration, the algorithm filters the searches that are valid candidates from the pool and selects the most promising ones.

A search is a valid candidate if and only if it is both *useful* and *searchable*. The search in the original search space is always useful. A search in an abstract search space is useful if and only if there are still states from the opposite frontier that do not correspond to a state already expanded in the abstract space. The main intuition behind this is that non-useful searches cannot possibly simplify the next step in the original search space. A search is *searchable* if the estimated time needed to perform the next step does not surpass the bounds imposed by our parameters. Among all the searches that are *valid candidates*, we select those that have a greater minimum f-value, because they are closer to proving that the current solution is optimal. If more than one search has the same minimum f-value, we select the one whose next step is estimated to take less time.

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, like in the standard implementation of BDDA*. If this was in the original state space, a new plan may be found (if the new plan has a lower cost, it is stored). If this was in an abstract state space, we update the heuristic value of the other searches in the opposite direction in the pool, both abstract and original. In order to easily re-evaluate the heuristic, we use the Symbolic List A* implementation proposed in (Edelkamp, Kissmann, and Torralba 2012).

If there are no valid search candidates (line 12), a new bidirectional search is added to the pool (which amounts to two new searches). First, we select a new abstraction strategy (line 13). Using the strategy, we relax the current frontiers of the original state space search, until the frontier size is small enough to continue the search and there is no previous equivalent search (line 14). Finally, the new search is included in the pool to be selected in subsequent iterations.

Abstraction Hierarchies

The SymBA* planner can be used with different abstraction hierarchies. We use the Symbolic M&S abstractions described in (Torralba, Linares López, and Borrajo 2013) and Symbolic PDB abstractions.

For the pattern selection, instead of choosing a particular fixed pattern for the abstraction like previous domain-independent methods (Haslum et al. 2007; Kissmann and Edelkamp 2011; Edelkamp 2006), we seek a complete hierarchy. This hierarchy consists of an ordering on the variables such that the variables at the beginning are abstracted first until a small enough abstraction is obtained. This ordering is similar to the one used by linear merge strategies of M&S, although in this case it is reversed, as the merge strategies consider important variables first whereas we abstract away important variables last.

These orderings are computed by adding goal variables to the ordering (Goal first strategies) or variables causally connected to variables already in the ordering (CG first strategies). Tie-breaking is performed either randomly (Random) or by taking Gamer's ordering (Gamer) or its reverse (Reverse Gamer). The SymBA* planner makes use of three different strategies, derived from the previously described criteria:

- CG Goal Random: Adds variables causally connected to

[2]This is not entirely true in the symbolic case, as having fewer states does not mean that the BDD that represents them is smaller, but in most cases there is a positive correlation.

variables already in the ordering. If there is none, a goal variable is added. Ties are broken randomly.

- Goal CG Gamer: Goal variables are added first. Then, variables causally connected to variables already in the ordering are added. Ties are broken by adding to the ordering variables lower in Gamer's ordering.
- Reverse Gamer: The ordering is the same as Gamer's, but reversed. This aims to obtain abstractions that are efficiently computable, as the variables are abstracted away from the top of the BDD. Goals and causal links are explicitly ignored, although they are taken into account in Gamer's ordering. Also, at some point the abstraction may not contain goal variables, although this is fine as long as we work with a perimeter.

The SymBA* Planner Configuration

The SymBA* planner is implemented on top of the Fast Downward planning system (Helmert 2006). We have presented two different configurations of SymBA* to the IPC-2014 competition: SymBA*-1 and SymBA*-2. They differ on the abstraction hierarchies used. They both use PDB abstractions, and SymBA*-2, additionally, uses M&S abstractions.

Each call to `Select-abstraction` returns a different abstraction, choosing an abstraction from each of the following hierarchies in a round robin schema:

1. (only in SymBA*-2) M&S using bisimulation shrinking, with a maximum number of 10000 abstract states
2. PDBs with CG Goal Random
3. PDBs with Goal CG Gamer
4. PDBs with Reverse Gamer

The strategies are always used in the same order. We bound the time available for selecting the abstraction to 500 seconds. Moreover, when the planner has spent 1500 seconds or 3GB, we consider that searching more abstractions to generate better heuristics will not pay off. In that case, the search continues only in the original state space, with the heuristics that have been generated so far.

All symbolic searches use the latest improvements on image computation (Torralba, Edelkamp, and Kissmann 2013) and h^2 constraints for symbolic search (Torralba and Alcázar 2013). In particular, we use a disjunctive partitioning of TRs with a maximum TR size of 100k nodes and a timeout of 60 seconds to generate the TRs. Constraints from the state invariants are encoded directly in the TRs.

Searches are considered to be searchable whenever their frontier has fewer than 10 million nodes and the next step is estimated to take at most 30 seconds. If the time bound is surpassed, the bound on the number of frontier nodes is updated to half of the current frontier. To guarantee that new abstract searches are searchable, the size of the frontier is reduced by abstracting away variables from the abstraction hierarchy until the relaxed frontier has fewer nodes than 80% of the bound on the number of nodes.

Preprocessing

There are two noteworthy considerations regarding the preprocessing phase in SymBA*:

- How the SAS^+ variables are selected.
- How to compute h^2 (Bonet and Geffner 2001) and prune spurious operators.

SAS^+ Variable Selection

Switching from Gamer's SAS^+ encoding to the Fast Downward version (Helmert 2009), we observed a decrease of performance in some benchmark domains. We changed the selection of which invariant groups are used as SAS^+ variables in order to avoid that degradation in performance.

The Fast Downward planner chooses invariant groups with the highest cardinality as SAS^+ variables, until all the fluents of the problem have been considered in a variable. Aiming to further reduce the number of SAS^+ variables selected, we prefer to select invariant groups that contain fluents that do not appear in other invariant groups. We base our criterion on the observation that, since all the fluents of the problem have to be included in a SAS^+ variable, invariant groups that have a fluent which does not appear in other invariant groups will always be selected anyway.

As an example, think of the following case, based on a simplified version of the IPC-2011 *floortile* domain: we have two robots on a grid such that the robots cannot be at the same cell at the same time. Two types of invariant groups are detected:

1. Each robot is at exactly one single cell:
   (at robot1 cell1),(at robot1 cell2),...
2. Each cell either is clear or has a robot at it:
   (clear cell1),(at robot1 cell1),(at robot2 cell1)

Invariants of the first type have larger cardinality, so Fast Downward would encode this problem with a variable per robot that represents the location of the robot ({*(at robot1 cell1),(at robot1 cell2),...*}) and a variable of the kind {*(clear cell1),⟨none of those⟩*} per cell. In our case, we prefer to select invariant groups of the second type first because each fluent *(clear cell1)* only takes part on a single invariant group. Thus, we would only have a variable per cell of the kind {(clear cell1),(at robot1 cell1),(at robot2 cell1)}, which amounts to fewer variables and fluents.

This leads to the use of *"exactly-one"* invariant groups as variables in most cases, avoiding the use of *"at-most-one"* invariant groups if possible – which require an additional *⟨none of those⟩* fluent. With this policy the number of resulting variables and fluents is usually lower. This may be counterproductive for techniques that depend on the causal graph, like the abstraction strategies that we use.

Computing h^2 Invariants and Pruning Spurious Operators

We have implemented the computation of the h^2 heuristic in Fast Downward's preprocessor. We also implemented a backward version of h^2 (Haslum 2008), which identifies

pairs of propositions that cannot be reached from goal states in regression.

We use the mutexes obtained from h^2 and the *"exactly-one"* invariant groups from Fast Downward's monotonicity analysis to disambiguate the preconditions and the effects of the operators of the problem (Alcázar et al. 2013). We discard operators whose preconditions or effects are spurious sets of fluents, that is, contradict the previously inferred state invariants. We do this because the number of ground operators is significantly reduced in many planning domains with respect to the standard preprocessor of Fast Downward.

The discovery and use of the state invariants during this phase is interleaved: whenever new mutexes or spurious operators are discovered in this process, we repeat the computation of h^2 in both directions and the operator disambiguation until no more constraints are inferred. We set a limit of 300 seconds for this phase.

Conditional Effect Support

The Fast Downward planning system partially supports conditional effects. They are correctly handled by the A^* search, but the current public version of M&S does not support them. We chose to disable M&S relaxations in these cases, so that only Symbolic Perimeter PDBs were used in domains with conditional effects.

To handle conditional effects in symbolic search, we encode them in the TRs. According to the semantics of conditional effects, they are applied in order. If more than one effect is applied over the same variable, the last one overwrites all the others. Hence, to generate the TR with conditional effects we group all the effects by the variable they modify. Each conditional effect is encoded as the conjunction of its condition, its effect and the negation of the conditions of previous effects over the same variable.

For the h^2 computation, conditional effects are not compiled away, but rather we ignore conditional preconditions and all the delete effects that are conditional or that delete a conditional effect.

Technical Details

To perform BDD operations, we used version 2.5 of Fabio Somenzi's CUDD library. The planner is compiled in 32-bit (-m32), using the compiler optimization (-O3) and with support of c++-11 features (–std=c++11).

Acknowledgements

This work has been partially supported by a FPI grant from the Spanish government associated to the MICINN project TIN2008-06701-C03-03, and it has also been supported by the project TIN2011-27652-C03-02.

References

Alcázar, V.; Borrajo, D.; Fernández, S.; and Fuentetaja, R. 2013. Revisiting regression in planning. In *International Joint Conference on Artificial Intelligence*, 2254–2260.

Alcázar, V.; Fernández, S.; and Borrajo, D. 2014. Analyzing the impact of partial states on duplicate detection and collision of frontiers. In *International Conference on Automated Planning and Scheduling*.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2):5–33.

Borrajo, D.; Kambhampati, S.; Oddi, A.; and Fratini, S., eds. 2013. *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. AAAI Conference on Artificial Intelligence (AAAI).

Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

de Champeaux, D. 1983. Bidirectional heuristic search again. *J. ACM* 30(1):22–32.

Edelkamp, S.; Kissmann, P.; and Torralba, Á. 2012. Symbolic A^* search with pattern databases and the merge-and-shrink abstraction. In Raedt, L. D.; Bessière, C.; Dubois, D.; Doherty, P.; Frasconi, P.; Heintz, F.; and Lucas, P. J. F., eds., *European Conference on Artificial Intelligence (ECAI)*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, 306–311. IOS Press.

Edelkamp, S. 2006. Automated creation of pattern database search heuristics. In *MoChArt*, 35–50.

Eyerich, P., and Helmert, M. 2013. Stronger abstraction heuristics through perimeter search. In Borrajo et al. (2013).

Felner, A., and Ofek, N. 2007. Combining perimeter search and pattern database abstractions. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, 155–168.

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 Conference on Artificial Intelligence (AAAI)*, 1007–1012. AAAI Press.

Haslum, P. 2008. Additive and reversed relaxed reachability heuristics revisited. *Proceedings of the 6th International Planning Competition*.

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*.

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

Helmert, M. 2009. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence* 173(5-6):503–535.

Holte, R. C.; Grajkowski, J.; and Tanner, B. 2005. Hierarchical heuristic search revisited. In *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., and Edelkamp, S. 2011. Improving cost-optimal domain-independent symbolic planning. In Burgard, W., and Roth, D., eds., *AAAI Conference on Artificial Intelligence (AAAI)*. AAAI Press.

Kwa, J. B. 1989. BS*: An admissible bidirectional staged heuristic search algorithm. *Artif. Intell.* 38(1):95–109.

Larsen, B. J.; Burns, E.; Ruml, W.; and Holte, R. 2010. Searching without a heuristic: Efficient use of abstraction. In *AAAI Conference on Artificial Intelligence (AAAI)*.

Leighton, M. J.; Ruml, W.; and Holte, R. C. 2011. Faster optimal and suboptimal hierarchical search. In *SOCS*.

McMillan, K. L. 1993. *Symbolic model checking*. Kluwer Academic publishers.

Pohl, I. S. 1969. *Bi-directional and Heuristic Search in Path Problems*. Ph.D. Dissertation, Stanford, CA, USA. AAI7001588.

Torralba, Á., and Alcázar, V. 2013. Constrained symbolic search: On mutexes, bdd minimization and more. In Helmert, M., and Röger, G., eds., *Symposium on Combinatorial Search (SoCS)*. AAAI Press.

Torralba, Á.; Edelkamp, S.; and Kissmann, P. 2013. Transition trees for cost-optimal symbolic planning. In Borrajo et al. (2013).

Torralba, Á.; Linares López, C.; and Borrajo, D. 2013. Symbolic merge-and-shrink for cost-optimal planning. In Rossi, F., ed., *International Joint Conference on Artificial Intelligence (IJCAI)*. IJCAI/AAAI.

New Encoding Methods for SAT-based Temporal Planning

Masood Feyzbakhsh Rankooh and **Gholamreza Ghassem-Sani**
Sharif University of Technology
Tehran, Iran

Abstract

Although satisfiability checking is known to be an effective approach in classical planning, it has scarcely been investigated in the field of temporal planning. Most notably, the usage of ∃-step semantics for encoding the problem into a SAT formula, while being demonstrably quite efficient for decreasing the size of the encodings in classical planning, has not yet been employed to tackle temporal planning problems. In this report, we describe ITSAT that uses temporal versions of classical ∀-step and ∃-step plans. We show that when the casual and temporal reasoning phases of a SAT-based temporal planner are separated, these semantics can be used to translate a given temporal planning problem into a SAT formula. We describe two different types of ∃-step encodings in temporal planning. The first encoding method is a temporal version of the classical ∃-step encoding. Like its classical counterpart, in the new encoding we suppose a few restrictive simplifying assumptions. On the other hand, by relaxing one of these assumptions, the second type of ∃-step encodings, which is often more compact than the first one, is explained. Our experiments indicate that by embedding the proposed encodings into ITSAT, a SAT-based temporal planner based on the ∀-step encoding, a considerable improvement is achieved in terms of both speed and memory usage of the planner.

Introduction

Previous research in the field of temporal planning has enormously benefited from employing well-developed classical planning strategies. In fact, many classical planning methods have already been used to tackle temporal planning problems, too. For instance, many successful temporal planners have utilized the ideas of partial order planning e.g., VHPOP (Younes and Simmons 2003) and CPT (Vidal and Geffner 2006). Planning graph analysis has also been adopted by temporal planners such as TGP (Smith and Weld 1999) and TPSYS (Garrido, Fox, and Long 2002). Some other temporal planners have embedded temporal reasoning into heuristic state space search. TFD (Eyerich, Matmuller, and Roger 2009) and POPF (Coles et al. 2010) are two successful instances of this latter approach.

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Employing satisfiability checking is another important trend of classical planning research. In this approach, a given planning problem is encoded into a SAT formula. In order to make the corresponding SAT formula finite, the potential plan is assumed to have a finite number of steps. The formula is given as the input to an off-the-shelf SAT solver. The SAT solver tries to find a model for the formula. If such a model exists, the final plan is extracted from it. Otherwise, the number of steps is increased and the whole process is repeated.

SAT-based classical planning was first used to find optimal plans, i.e., plans with minimum number of actions (Kautz and Selman 1992). To guarantee the optimality of the output plan, the formulae must include certain clauses to ban each step from containing more than one action. However, if optimality is not the objective of the planner, forcing single-action steps is not necessary. In an alternative approach, which has been shown to be quite effective (Rintanen, Heljanko, and Niemelä 2006), the planning problem is encoded in such a way that each step of the final plan can have several parallel actions. The usage of multiple-action steps results in a smaller number of steps in SAT formulae, which in turn reduces the number of SAT variables. Since the speed of SAT solvers may exponentially decrease as the number of variables is increased, employing this idea often results in considerably faster planners. Several encoding methods have been introduced to take advantage of such a parallelism. The research in this area is mainly focused on the so-called ∀-step and ∃-step semantics of valid plans (Rintanen, Heljanko, and Niemelä 2006).

The ∀-step semantics allows each step of a plan to include a particular set of actions, only if those actions can be executed in every possible order without affecting the validity of the plan. On the other hand, the ∃-step semantics is based on some weaker requirements: for each step of a plan, there must exist at least one possible ordering in which the actions of that step can be arranged without falsifying the validity of the plan. It should be clear that the ∃-step semantics potentially allows more parallelism than what is permitted by the ∀-step semantics. In fact, ∃-step encoding has been shown to be one of the most efficient methods for converting classical planning problems to SAT formulae (Rintanen, Heljanko, and Niemelä 2006).

Satisfiability checking has also been used to tackle tem-

poral planning problems. STEP (Huang, Chen, and Zhang 2009) and T-SATPLAN (Mali and Liu 2006) are two SAT-based planners that handle temporal constraints by assigning explicit discrete time labels to each step of the encoding. TM-LPSAT (Shin and Davis 2005), which has been designed to solve planning problems defined by PDDL+ (Fox and Long 2002), is another SAT-based planner that can handle temporal planning problems. However, in TM-LPSAT, the steps of the SAT formula do not possess predefined time labels. Instead, the execution time of each step will be stored in a variable whose value is to be determined by an SMT solver (Armando and Giunchiglia 1993).

ITSAT (Rankooh, Mahjoob, and Ghassem-Sani 2012) is yet another example of SAT-based temporal planners. Like TM-LPSAT, ITSAT does not assign explicit time labels to the steps of its encoding. Besides, ITSAT first abstracts out the durations of actions. It then finds a plan that is only causally valid. The plan is then refined in such a way that satisfies temporal constraints imposed by the durations of actions.

The latest version of ITSAT generalizes the concepts of single-action-step, $\forall$-step, and $\exists$-step plans to the temporal planning context. In this reporst, we show that according to our definition of parallel plans, STEP, T-SATPLAN, and TM-LPSAT are all using a temporal version of the single-action-step encoding. We also show that the separation of causal and temporal reasoning has enabled ITSAT to somehow use the temporal version of $\forall$-step semantics for its encoding. We also describe a temporal version of $\exists$-step semantics used by ITSAT in its encoding phase.

Preliminaries

In this section, we define basic concepts such as temporal states, actions, problems, and plans. Our definitions of these concepts here are consistent with the level 3 of PDDL2.1 (Fox and Long 2003), and have been inspired by the formalization used for TEMPO (Cushing et al. 2007). We assume that the reader is familiar with the definitions of states, actions, and problems of classical planning.

Definition 1 (temporal states) A temporal state, s, is a pair $(state(s), agenda(s))$, where $state(s)$ is a classical planning state and $agenda(s)$ contains all the actions that are started but not yet finished before reaching s.

Definition 2 (temporal actions and events) A temporal action, a, is a quadruple $(start(a), end(a), over(a), dur(a))$ where $start(a)$ and $end(a)$ are two classical planning actions denoting the starting and ending events of a, $over(a)$ is a set of classical preconditions representing the over-all conditions of a, and $dur(a)$ is a positive rational number specifying the duration of a.

Definition 3 (mutual exclusion) Two events, e_i and e_j, are mutually exclusive in the temporal sense if either of the following conditions holds:

- e_i and e_j are mutually exclusive in the classical sense (Blum and Furst 1997).
- e_i (or e_j) is the staring event of action a, and e_j (or e_i) deletes a member of $over(a)$.

Definition 4 (applicability) A set of events, $E = \{e_1, ..., e_n\}$, is applicable in state s, if all following conditions hold:

- For each i, e_i is applicable to $state(s)$ in the classical sense.
- If e_i is the starting event of action a, then $over(a) \subseteq \bigcup_{e \in E} add(e) \cup state(s)$
- If e_i is a starting event, then it does not delete an over-all condition of any member of $agenda(s)$.
- If e_i is the ending event of action a, then a is a member of $agenda(s)$ and e_i does not delete an over-all condition of any other member of $agenda(s)$.
- For all i and j, e_i and e_j are not mutually exclusive.

We say that members of E are simultaneously applied to state s.

Definition 5 (successors) If a set of events $E = \{e_1, ..., e_n\}$ is applicable to state s, it will change s to s' where $state(s')$ is the result of applying all members of E to $state(s)$ in the classical sense and in an arbitrary order, and $agenda(s')$ is determined by the following rule: $agenda(s') = agenda(s) \cup \{a | start(a) \in E\} - \{a | end(a) \in E\}$. s' is also denoted by $succ(s, E)$. Applying a sequence of sets of event to step s is defined by the following recursive rule: $succ(s, \langle E_1, ..., E_n \rangle) = succ(succ(s, E_1), \langle E_2, ..., E_n \rangle)$.

Definition 6 (temporal problems) A temporal problem, P, is a triple (I, G, A) where I is a temporal state such that $agenda(I) = \phi$ representing the initial state, G is a set of classical goal conditions, and A is the set of all possible temporal actions of P.

Definition 7 (temporal plans) A temporal plan π is a sequence $E_1, ..., E_n$, where each E_i is a set of simultaneously executed events representing a step of π. π is valid for problem $P = (I, G, A)$ if there exist a sequence $s_0, ..., s_n$ of temporal states, such that $s_0 = I$, $G \subseteq state(s_n)$, $agenda(s_n) = \phi$, and for every i, $s_i = succ(s_{i-1}, E_i)$. Moreover, there must exist a scheduling function $\tau : \{1, ..., n\} \rightarrow$ Q with the following properties:

- For all i, $\tau(i) < \tau(i+1)$.
- For each $a \in A$, if $start(a) \in E_i$ and $end(a) \in E_j$, then $\tau(j) = \tau(i) + dur(a)$.

It should be noted that by Definition 7, a valid temporal plan is a sequence of steps where each step includes several simultaneously executed events. In other words, all events of any particular step must be executed at the same time. We call this semantics, 1-step semantics for temporal plans. This is in fact a generalization of classical single-action-step semantics.

Simultaneity of events is necessary for solving some temporal problems. For instance, consider the plan shown in Figure 1(a). In this plan, we have two temporal actions a

and b, where the starting event of each is providing the over-all conditions of the other. Consequently, if the goal state is reached by either a or b, both actions have to be started simultaneously. Another example of situations where simultaneity of events is necessary is depicted in Figure 1(b). In this example, the over-all condition of a is added and deleted respectively by the starting and ending events of b. The fact that a and b have equal durations necessitates the simultaneous execution of these two actions.

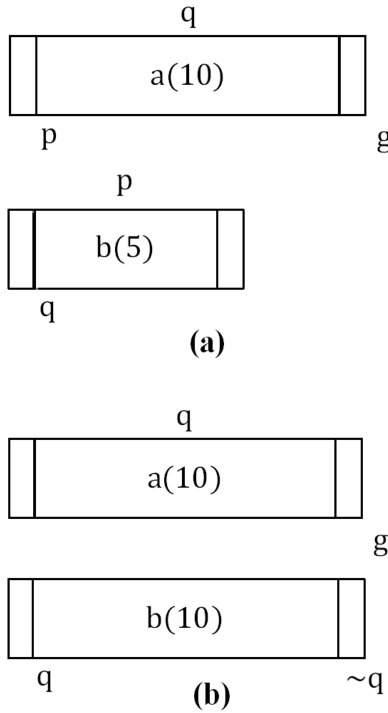


Figure 1. Plans with simulataneous events

ITSAT Planning System

In this section, an older version of ITSAT planning system is briefly described. ITSAT was the first SAT-based temporal planner in which the causal and temporal reasoning tasks were performed in two separate phases. Such a separation is critical for the feasibility of our ∃-step encoding methods discussed later.

To solve a temporal planning problem, ITSAT first abstracts out the durations of actions. In other words, it is assumed that actions can have arbitrary durations. It then encodes the abstract problem into a SAT formula. This abstraction causes the encoding to be very similar to that of classical SAT-based planners. However, beside ordinary clauses used by classical SAT-based planners, ITSAT needs a number of extra clauses to satisfy the over-all conditions of actions. Moreover, there are several clauses to appropriately manipulate the agendas of states before and after each step.

By using the encoding method explained above, ITSAT may find plans that are not temporally valid. However, all the obtained plans are guaranteed to be what we call *causally valid*.

Definition 8 (causally valid temporal plans) A temporal plan π is causally valid for temporal problem P, if it admits all requirements of definition 7, except for the existence of the scheduling function τ, which is optional.

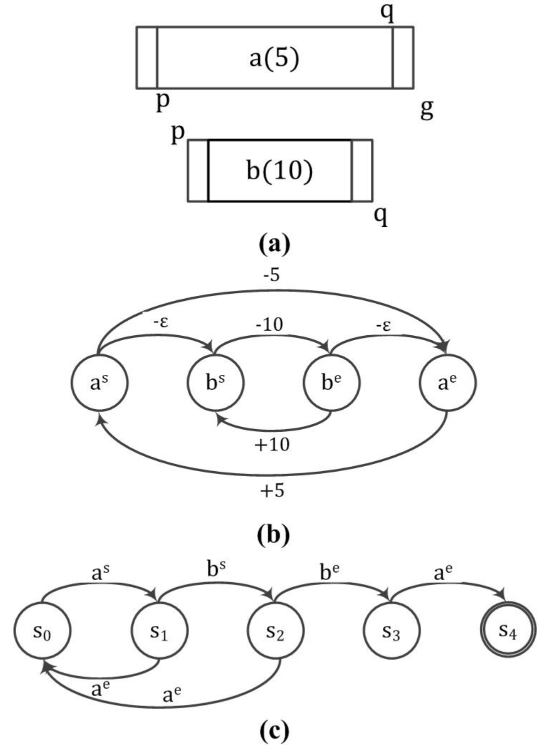


Figure 2. Negative cycle detection in ITSAT

Figure 2(a) represents a causally valid temporal plan that is not temporally valid. That is because action b is to be executed during the execution of action a, while the duration of b is greater than that of a.

To find a temporally valid plan, ITSAT tries to schedule the events of the obtained causally valid plan by solving a particular Simple Temporal Problem (STP) (Dechter, Meiri, and Pearl 1991) that enforces the temporal constraints of the planning problem. If the STP is inconsistent, there must exist a negative cycle in the corresponding Simple Temporal Network (STN). The STN of the plan in Figure 2(a) is shown in Figure 2(b), where the cycle $a^s b^s b^e a^e a^s$ is a negative cycle. The sequence of events that lead to such negative cycles can be detected by simple Finite State Machines (FSMs). The transition of these FSMs can then be turned into appropriate caluses that collectively prevent such negative cycles from reoccurring. The FSM that detects the cycle $a^s b^s b^e a^e a^s$ is depicted in Figure 2(c). It has been shown that ITSAT is capable of solving problems with the required concurrency property (Cushing et al. 2007), and is competitive with the state-of-the-art temporally expressive planners.

Semantics for Causally Valid Temporal Plans

According to definition 7, although planners such as STEP, T-SATPLAN, and TM-LPSAT allow parallel execution of actions, they are in fact using the 1-step encoding. That is because these planners assume simultaneous execution of all events in each step.

As we mentioned before, the classical $\forall$-step semantics permits the execution of more than one action in each step, only if the validity of the plan is not dependent on the execution order of those actions. This can simply be guaranteed by adding a particular clause for each pair of mutually exclusive actions to ensure that those actions will not be included in the same step. However, such a strategy does not work for temporal planning. In temporal planning, because of the temporal constraints between the starting and ending events of actions, the validity of a particular ordering of events of a step, also depends on the ordering of events of other steps. Nevertheless, in ITSAT this problem has been tackled by separating the causal and temporal reasoning phases. In general, if we focus on finding causally valid plans, and postpone the scheduling phase, the mentioned problem about checking the feasibility of imposing different orderings of events in each step will no longer exist.

We now describe our semantics for causally valid $\forall$-step and $\exists$-step temporal plans.

Definition 9 (temporal $\forall$-steps and $\exists$-steps) Let $S = \{E_1, ..., E_n\}$ be a set of sets of events, and s_1 and s_2 be two temporal states. S is a temporal $\forall$-step from s_1 to s_2 if for all one-to-one ordering functions $O : \{1, ..., n\} \rightarrow \{1, ..., n\}$, we have: $s_2 = succ(s_1, \langle E_{O(1)}, ..., E_{O(n)} \rangle)$.
S is a temporal $\exists$-step from s_1 to s_2 if for some one-to-one ordering functions $O : \{1, ..., n\} \rightarrow \{1, ..., n\}$, we have: $s_2 = succ(s_1, \langle E_{O(1)}, ..., E_{O(n)} \rangle)$

Definition 10 (causally valid $\forall$-step and $\exists$-step temporal plans) Let $P = (I, G, A)$ be a temporal planning problem. Suppose $s_0, ..., s_n$ is a sequence of temporal states such that $s_0 = I$, $G \subseteq state(s_n)$, and $agenda(s_n) = \phi$. If for each $1 \leq i \leq n$, $Step_i$ is a $\forall$-step ($\exists$-step) from s_{i-1} to s_i, then we call the sequence $\langle Step_1, ..., Step_n \rangle$, a causally valid $\forall$-step ($\exists$-step) temporal plan for P.

$\exists$-step Encodings for Causally Valid Temporal Plans

Classical $\exists$-step encoding, which has been introduced in (Rintanen, Heljanko, and Niemelä 2006), is based on the $\exists$-step semantics for classical valid plans. However, for the sake of improving the efficiency of the planner, the following restrictive rules have been also enforced on it.

- Rule 1: Instead of accepting all possible orderings among the actions of each step, only a fixed arbitrary ordering was allowed. Executing a step means executing its actions according to this fixed ordering.
- Rule 2: Preconditions and effects of all actions of each step must be consistent with the states before and after that step, respectively.

The second rule causes an action a to be excluded from a step if there is a contradiction between its effects and that of any other action in that step. Also, a is prevented from being in a step if its precondition is deleted in that step by any other action that according to the predefined fixed ordering is located before a.

In this section, we explain two $\exists$-step encodings for temporal planning. Both proposed encodings are based on the $\exists$-step semantics for causally valid temporal plans (definition 10). By considering events, instead of actions, both rules mentioned above can be applied to temporal planning, too. While in our first encoding, we respect both rules, our second encoding relaxes the second one. Besides, we also use a third restrictive rule in both proposed encodings. We will later discuss the benefits of the third rule.

- Rule 3: The ending event of each action must be located next to the starting event of that action in the fixed ordering mentioned in Rule 1.

It should be noted that since we are using a total order between all events, our encodings are not completely coherent with the $\exists$-step semantics defined by definition 10. In fact, for the sake of simplicity, we have assumed that no pair of actions can happen simultaneously in a causally valid plan. This assumption does not render our encodings incomplete unless the problem has a certain property that we call *required causal simultaneity*.

Definition 11 (required causal simultaneity) We say a temporal plan $\pi = \langle E_1, ..., E_n \rangle$ has simultaneity if for some i, we have $|E_i| > 1$. If every causally valid plan of a temporal problem P has simultaneity, we say that P requires causal simultaneity.

Note that while the plan in Figure 1(a) requires causal simultaneity, this is not the case in the plan presented in Figure 1(b). Moreover, while required simultaneity entails required concurrency (Cushing et al. 2007), the reverse is not true. In other words, required simultaneity is more specific than required concurrency. We now show that the existence of the required causal simultaneity has some necessary (but not sufficient) conditions that can be detected in polynomial time.

Let P be a temporal planning problem. Associated with P, we construct a precedence graph $G(P) =< V, E >$ as follows:

- For each event e_i of P, there is a vertex $v_i \in V$.
- If e_i is the starting event of action a, and e_j adds an over-all condition of a, we add a directed edge (v_j, v_i) to E.
- If e_i is the ending event of action a, and e_j deletes an over-all condition of a, we add a directed edge (v_i, v_j) to E.

The precedence graphs of the problems corresponding to the plans of Figure 1(a) and Figure 1(b) are presented in Figure 3(a) and Figure 3(b), respectively.

Theorem 1. Let P be a temporal planning problem for which there exist a causally valid temporal plan. If P requires causal simultaneity, then $G(P)$ must have a cycle.

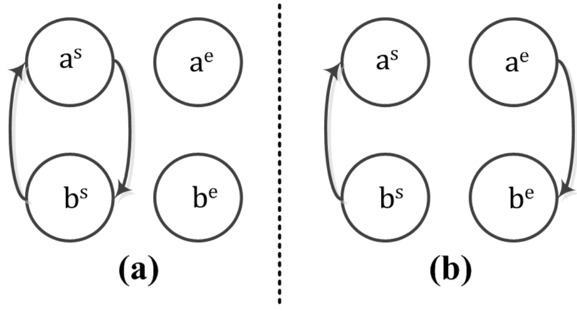


Figure 3. Precedence graphs

Proof sketch. The proof is given by contradiction. Suppose that $G(P)$ is acyclic. By applying the topological sort algorithm to $G(P)$, we obtain a total ordering on the vertices of (and consequently on the events of P). Let π be a causally valid temporal plan for P. We construct a new plan π', which is the same as π except for the previously simultaneous events that are now totally ordered by the topological ordering. The ordering we imposed on the events of π' prevents it from becoming causally invalid (details are omitted here). This contradicts our assumption that P requires causal simultaneity. □

Since topological sort is a polynomial time algorithm, we conclude that detecting the necessary conditions of required causal simultaneity, stated in theorem 1, can be done in polynomial time. Our investigations show that from all domains used in different International Planning Competitions, only the rovers domain has this necessary condition. Therefore, preventing the occurrence of simultaneous events in the causally valid plans will not seriously damage the generality of our method.

We now describe the clauses that are to be included in both new encodings. These are clauses needed for appropriate manipulation of the agendas of states and preventing the over-all conditions of each action from being deleted during the execution of that action.

Clauses Shared by Both Proposed Encodings

Assume that the encoding is to represent a $\exists$-step temporal plan, $\langle Step_1, ..., Step_n \rangle$, where members of each $Step_t$ are applied to the temporal state s_{t-1}, according to a predefined fixed ordering, and map it to state s_t. Suppose that the index of each event represents the location of that event in the fixed ordering. From now on, if we say that an event e_i is earlier (later) than an event e_j, we mean that j is greater (less) than i. We denote the existence of an event e in a step t by the SAT variable Y_e^t. We also use the SAT variable O_a^t to denote that action a is a member of $agenda(s_t)$. The SAT variable X_p^t is used to denote the existance of proposition p in state s_t.

Encoding the initial state and goal conditions of the problem is quite standard. The following clauses are introduced to guarantee that the agendas of states are changed appropriately. We give both a verbal description, and a formal representation of each clause.

- If e_i is the starting event of action a, the presence of e_i in step t implies that a is not a member of $agenda(s_{t-1})$. Besides, if a is a member of $agenda(s_t)$ but not $agenda(s_{t-1})$, then e_i must be present in step t: $(Y_{e_i}^t \rightarrow \sim {O_a}^{t-1})$ and $(\sim O_a^{t-1} \wedge O_a^t \rightarrow Y_{e_i}^t)$. Furthermore, if e_i is present in step t, but e_{i+1}, which according to our third restrictive rule must be the ending event of a, is not present in step t, then a has to be a member of $agenda(s_t)$: $Y_{e_i}^t \wedge \sim Y_{e_{i+1}}^t \rightarrow O_a^t$.
- A description analogous to what is given above also applies to the ending event of a: $(Y_{e_{i+1}}^t \rightarrow \sim {O_a}^t)$, $(O_a^{t-1} \wedge \sim O_a^t \rightarrow Y_{e_{i+1}}^t)$, and $(\sim Y_{e_i}^t \wedge Y_{e_{i+1}}^t \rightarrow O_a^{t-1})$.
- The agenda of the initial and final state of the plan must be empty: for each a, $(\sim O_a^0 \wedge \sim O_a^n)$.

The following clauses are added to the encoding for preventing the over-all conditions of each action from being deleted during the execution of that action. These clauses are representing a schematic message passing strategy, which is inspired by the chaining method used in (Rintanen, Heljanko, and Niemelä 2006). For each step t, proposition p, and event e_i whose corresponding action has p as an over-all condition, the SAT variable $B_{p,i}^t$ denotes whether or not p is deleted in step t, by an event whose index is less than i (i.e., an earlier event). Similarly, variable $A_{p,i}^t$ represents that whether or not p is deleted in step t by an event whose index is greater than i (a later event). Finally, variable D_p^t shows whether or not p is deleted by any event in step t.

- Assume that event e_i deletes proposition p, and e_j is the first event after e_i with the property that its corresponding action has p as an over-all condition. If e_i is present in step t, then a message must be sent to e_j to indicate that p has been deleted earlier in step t: $(Y_{e_i}^t \rightarrow B_{p,j}^t)$ and $(Y_{e_i}^t \rightarrow D_p^t)$.
- Assume that event e_i deletes proposition p, and e_j is the last event before e_i with the property that its corresponding action has p as an over-all condition. If e_i is present in step t, then a message must be sent to e_j to indicate that p will be deleted later in step t: $(Y_{e_i}^t \rightarrow A_{p,j}^t)$.
- Assume that p is an over-all condition of action a, e_i is the ending event of a, and e_j is the first ending event after e_i with the property that its corresponding action has p as an over-all condition. If e_i receives a message implying that p has been deleted earlier in step t, then it must pass this message to e_j: $(B_{p,i}^t \rightarrow B_{p,j}^t)$.
- Assume that p is an over-all condition of action a, e_i is the starting event of a, and e_j is the last starting event before e_i with the property that its corresponding action has p as an over-all condition. If e_i receives a message implying that p is to be deleted later in step t, then it must pass this message to e_j : $(A_{p,i}^t \rightarrow A_{p,j}^t)$.
- Assume that p is an over-all condition of action a. If p is deleted in step t, then a cannot be a member of both $agenda(s_{t-1})$ and $agenda(s_t)$: $(D_p^t \wedge O_a^t \rightarrow \sim O_a^{t-1})$. In other words, if a is started before and ended after step t, its overall conditions cannot be deleted in step t.
- Assume that p is an over-all condition of action a, and e_i is the starting event of a. If e_i is present in step t, and p

has been deleted by an event later than e_i in step t, then e_{i+1} (the ending event of a) must be present in step t, too: $(Y^t_{e_i} \wedge A^t_{p,i} \rightarrow Y^t_{e_{i+1}})$. This implies that if a is started but not ended in step t, its over-all condition cannot be deleted later in step t. This is where we are taking advantage of our third restrictive rule: if a is both started and ended in the same step, because its starting and ending events are next to each other, no other event can delete its over-all conditions while a is being executed.

- Assume that p is an over-all condition of action a, and e_i is the ending event of a. If e_i is present in step t, and p is deleted by an event earlier than e_i in step t, then e_{i-1} (i.e., the starting event of a) must be present in step t, too: $(Y^t_{e_i} \wedge B^t_{p,i} \rightarrow Y^t_{e_{i-1}})$. This implies that if a is ended but not started in step t, its over-all condition cannot be deleted in step t by an earlier event.

In addition to the shared clauses stated above, there are other necessary clauses exclusive to each of our new encodings. We present these clauses in their corresponding subsection. We say that an event e requires a proposition p if p is a precondition of e, or e is the starting event of an action that has p as an over-all condition. We say that the a proposition p is relevant to an event e if e requires, adds, or deletes p.

A Natural Extension to the Classical ∃-step Encoding

Our first proposed encoding is a natural extension to the classical ∃-step encoding, as it uses all three restrictive rules stated above. According to the second rule, the preconditions and effects of events of each step must be consistent with the states before and after that step, respectively. Therefore, this part of the encoding, which also includes explanatory frame axioms, is very similar to its corresponding part in the standard classical encodings. However, when we are dealing with temporal planning problems, the over-all conditions of actions must be encoded, too:

- Assume that e is the starting event of an action a, p is an over-all condition of a, and e does not add p. If e is present in any step t, then p must be true in s_{t-1}: $(Y^t_e \rightarrow X^{t-1}_p)$.

Similar to the classical ∃-step encoding, for any $i < j$, if e_i deletes p and e_j requires p, then e_i and e_j cannot be both present in any step. To ensure this, a schematic message passing strategy very similar to the one mentioned before, is employed.

For each step t, proposition p, and event e_i that requires p, $Z^t_{p,i}$ denotes if p is deleted in step t by an event earlier than e_i.

- Assume that e_i deletes p, and e_j is the first event after e_i with the property of requiring p. If e_i is present in step t, then a message must be sent to e_j to indicate that p has been deleted earlier in step t: $(Y^t_{e_i} \rightarrow Z^t_{p,j})$.
- Assume that e_i requires proposition p, and e_j is the first event after e_i with the property of requiring p. If e_i receives a message implying that p has been deleted earlier in step t, then e_i cannot be present in step t, and it must pass this message to e_j: $(Z^t_{p,i} \rightarrow\sim Y^t_{e_i} \wedge Z^t_{p,j})$.

Relaxed ∃-step Encoding

The second restrictive rule presented before, prevents a proposition from being both produced and used in the same step of the final plan. It also does not allow the deletion and production of any particular proposition to happen in the same step. By relaxing these restrictions the encodings can be further compressed, i.e., the relaxation permits more events in each steps. In classical planning, a less relaxed form of Rule 2 has been used in (Wehrle and Rintanen 2007), where the effects of actions in each step can be used by other actions in that step. However, here we totally relax Rule 2 and allow each proposition to be required, added, and deleted in each step as many times as is needed. In the first encoding, explained in the previous subsection, we inform events if any of their requirements is deleted earlier in the same step. In the relaxed encoding, however, the events are informed about the very last change in the truth value of their requirements. No event can occur in the final plan unless the last change in the truth value of any of its requirements has caused the requirement to become true. For each step t, event e_i, and proposition p that is relevant to e_i, $V^t_{p,i}$ represents the truth value of p just before the hypothetical execution of e_i.

- Assume that e_i deletes p, and e_j is the first event after e_i with the property of having p as a relevant proposition. If e_i is present in step t, then a message must be sent to inform e_j that the last change in p has been performed to delete it: $(Y^t_{e_i} \rightarrow\sim V^t_{p,j})$. An analogous discussion is valid when e_i adds p: $(Y^t_{e_i} \rightarrow V^t_{p,j})$. Moreover, if e_i is not present in step t, it must pass any received information regarding the value of p to e_j: $(\sim Y^t_{e_i} \wedge V^t_{p,i} \rightarrow V^t_{p,j})$ and $(\sim Y^t_{e_i} \wedge \sim V^t_{p,i} \rightarrow\sim V^t_{p,j})$.
- Assume that e_i is the last event in the ordering with the property of having p as a relevant preposition. If e_i is not present in step t, the value of p just before e_i must be transferred to the next step: $(\sim Y^t_{e_i} \wedge V^t_{p,i} \rightarrow X^t_p)$ and $(\sim Y^t_{e_i} \wedge \sim V^t_{p,i} \rightarrow\sim X^t_p)$. Moreover, If e_i deletes p we add the clause $(Y^t_{e_i} \rightarrow\sim X^t_p)$ to ensure that the presence of e_i in step t, implies that p is not true in s_t. Again, an analogous discussion is valid when e_i adds p: $(Y^t_{e_i} \rightarrow X^t_p)$.
- Assume that e_i is the first event in the ordering with the property of having p as a relevant proposition. e_i must be informed of the value of p in s_{t-1}: $(X^{t-1}_p \leftrightarrow V^t_{p,i})$.
- Assume that e_i requires p. If e_i is present in step t, then p must be true just before the execution of e_i: $(Y^t_{e_i} \rightarrow V^t_{p,i})$.

Implementation Details and Empirical Results

We have incorporated our new encoding methods into the older verrsion of ITSAT. ITSAT has been slightly modified so that our new encodings can coherently work with it. These modifications have been applied to mutual exclusion analysis and negative cycle detection parts of ITSAT.

It is known that SAT-based planners can significantly benefit from inference about mutually exclusive propositions (Kautz, Selman, and Hoffmann 2006). Any two propositions

that remain mutually exclusive after the planning graph has been leveled off (Blum and Furst 1997), cannot be both true in the same state of a valid plan. These are the only mutual exclusion relations that are included in our new encodings.

The second minor modification of ITSAT is related to its FSMs that detect certain negative cycles. As we mentioned before, the negative cycles are prevented by encoding the transitions of a particular FSM to the SAT formula. In our ∃-step encodings, the events of each step are assumed to be executed according to a predefined ordering. We have slightly modified the encoding of ITSAT to impose the corresponding order on the transitions of each FSM in each step.

For evaluating the proposed encoding methods, we have tested three versions (i.e., the original ∀-step, the ∃-step, and the relaxed ∃-step version) of ITSAT on the problem sets of previous International Planning Competitions. The experiments have been conducted on a 3.1GHz corei5 CPU with 4GB main memory. Precosat (Biere 2009), which is a free off-the-shelf SAT solver, has been used for satisfying SAT formulae in all three versions of ITSAT. For each problem and each version of ITSAT, several SAT formulae with increasing number of steps were produced. Some of the results are shown in Table 1.

The columns of Table 1 represent: the name of the domain, the problem number, the used encoding method, the number of steps in the encoding, the result of precosat in terms of satisfiability or unsatisfiability of the formula, the number of clauses and variables divided by 1000, the amount of time taken by precosat to determine the result, and the amount of memory needed for saving the formula. For each problem and each encoding method, the results are presented for two cases: unsatisfiable formula with the highest number of steps, and satisfiable formula with the lowest number of steps. We have used symbol $\exists^*$ as an abbreviation for the relaxed ∃-step encoding. Symbol ∞ is used in the time column for cases in which precosat has not determined the satisfiability of the formula in 1800 seconds. Please note that in the sokoban domain, an extra comparison between $\exists^*$ and ∃ configurations has been presented in Table 1 for problem no. 1 in which the ∀ configuration could not find a plan.

As it is shown in Table 1, using our proposed encodings causes a considerable improvement in terms of both speed and memory usage of the planner. Furthermore, the relaxed ∃-step encoding is faster than the other two in almost all domains. Although the results presented in table 1, does not cover all the domains used in previous IPCs, we should mention that the same pattern has been observed in nearly all of those domains, too.

domain	prob	enc	steps	res	$\frac{C}{1000}$	$\frac{V}{1000}$	time (s)	mem (MB)
sokoban (2011)	4	∀	29	F	1907	60	∞	134
		∃	13	F	349	86	7	45
		$\exists^*$	6	F	219	101	11	62
		∀	30	T	1973	63	465	138
		∃	14	T	378	93	12	47
		$\exists^*$	7	T	256	118	11	64
	1	∃	12	F	765	187	233	95
		$\exists^*$	4	F	346	157	7	70
		∃	13	T	835	203	120	99
		$\exists^*$	5	T	435	197	48	124
parcprint. (2011)	12	∀	52	F	8080	153	10	509
		∃	31	F	1442	220	1.4	134
		$\exists^*$	13	F	689	74	1.4	139
		∀	53	T	8233	156	8	518
		∃	32	T	1929	293	2.1	137
		$\exists^*$	14	T	744	83	1.7	142
floortile (2011)	10	∀	30	F	250	22	139	21
		∃	11	F	67	18	0.5	10
		$\exists^*$	8	F	69	27	1.3	16
		∀	31	T	257	22	169	21
		∃	12	T	74	20	0.7	11
		$\exists^*$	9	T	78	31	1.1	17
crewplan. (2011)	1	∀	41	F	80	10	∞	8
		∃	41	F	59	17	∞	10
		$\exists^*$	9	F	241	74	0.7	39
		∀	42	T	83	10	2.9	8
		∃	42	T	60	18	2.2	10
		$\exists^*$	10	T	269	83	0.7	45
pegsol (2011)	20	∀	26	F	73	6	∞	7
		∃	12	F	23	6	∞	4
		$\exists^*$	5	F	15	7	0	4
		∀	27	T	77	6	5	8
		∃	13	T	25	7	11	6
		$\exists^*$	6	T	19	8	0.4	4
depots (2004)	10	∀	9	F	25155	145	6	1533
		∃	5	F	864	215	1.3	97
		$\exists^*$	4	F	977	483	2	250
		∀	10	T	28741	163	11	1745
		∃	6	T	1058	260	4.7	103
		$\exists^*$	5	T	1237	607	4	266
driverlog (2004)	15	∀	17	F	15365	217	15	939
		∃	8	F	739	199	2.3	97
		$\exists^*$	6	F	647	246	1	187
		∀	18	T	16606	232	9	1058
		∃	9	T	841	225	8	104
		$\exists^*$	7	T	763	266	2	240

Table 1. Comparing Different Encoding Methods

Conclusion

In this paper, we explained ITSAT planner that by separating the casual and temporal reasoning phases of a SAT-based temporal planner, can employ compact semantics to construct effective encodings. Two different types of ∃-step encodings were described for temporal planning. We have

embedded our new ∃-step encodings into ITSAT. We empirically showed the new encodings to be more efficient than the ∀-step encoding employed previously in ITSAT.

References

Armando, A.; and Giunchiglia E. 1993. Embedding Complex Decision Procedures inside an Interactive Theorem Prover. *Annals of Mathematics and Articial Intelligence*, 8(34), 475502.

Biere, A. 2009. P{re,i}coSAT@SC'09. Solver description for SAT Competition 2009. *In SAT 2009 Competitive Event Booklet*.

Blum, A.; and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial intelligence*. 90:281-300.

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order Planning. *Proceedings of 20th International Conference on Automated Planning and Scheduling*, 42-49, AAAI press.

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. S. 2007. When is temporal planning really temporal? *Proceedings of 20th International Joint Conference on Artificial Intelligence*, 1852-1859, AAAI press.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence* 49(1-3): 61-95.

Eyerich, P.; Mattmuller, R.; and Roger, G. 2009. Unifying Context-Enhanced Additive Heuristic for Temporal and Numeric Planning. *Proceedings of 19th International Conference on Automated Planning and Scheduling*, AAAI press.

Fox, M.; and Long, D. 2002. PDDL+: Modelling Continuous Time-dependent Effects. *In Proceedings of the Third International NASA Workshop on Planning and Scheduling for Space*.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20: 61-124.

Garrido, A.; Fox, M.; and Long, D. 2002. A temporal planning system for durative actions of PDDL2.1. *Proceedings of 15th European Conference on Artificial Intelligence*, 586-590, IOS press.

Huang, R.; Chen, Y.; and Zhang, W. 2009. An optimal temporally expressive planner: Initial results and application to P2P network optimization. *Proceedings of 19th International Conference on Automated Planning and Scheduling*, AAAI press.

Kautz H.; and Selman, B. 1992. Planning as Satisfiability. *Proceedings of 10th European Conference on Artificial Intelligence*, 359-363, IOS press.

Kautz, H.; Selman, B.; and Hoffmann, J. 2006. SatPlan: Planning as Satisfiability. *International Planning Competition*.

Mali, A. D.; and Liu, Y. 2006. T-SATPLAN: A SAT-based Temporal Planner. *International Journal of Artificial Intelligence Tools* 15(5): 779-802.

Rankooh, M. F.; Mahjoob, A.; and Ghassem-Sani, G. 2012. Using Satisfiability for Non-Optimal Temporal Planning. *Proceedings of the 13th European Conference on Logics in Artificial Intelligence*, Springer.

Rintanen, J.; Heljanko, K.; and Niemelä. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13): 1031-1080.

Shin, J.; and Davis, E. 2005. Processes and continuous change in a SAT-based planner. *Artificial Intelligence*, 166(1-2): 194-253.

Smith, D. E.; and Weld D. S. 1999. Temporal planning with mutual exclusion reasoning. *Proceedings of 16th International Joint Conference on Artificial Intelligence*, 326-337, AAAI press.

Vidal, V.; and Geffner, H. 2006. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence*, 170(3): 298-335.

Wehrle, M.; and Rintanen, J. 2007. Planning as Satisfiability with Relaxed ∃-step Plans. *Proceedings of 20th Australian Joint Conference on Artificial Intelligence*, 244-253, Springer-Verlag.

Younes, H. L. S.; and Simmons, R. G. 2003. VHPOP: Versatile Heuristic Partial Order Planner. *Journal of Artificial Intelligence Research*, 20: 405-430.

tBurton: A Divide and Conquer Temporal Planner

David Wang and **Brian Williams**
Model-based Embedded and Robotic Systems Group
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract

tBurton is a temporal planner designed to solve model-based control problems involving time. Unlike purely heuristic-based planning strategies, tBurton first uses causal graph decomposition to divide the problem into factors, and then plans for each factor using an off-the-shelf heuristic forward search planner. These 'sub-plans' are then re-combined in a conflict directed search which also learns sub-plans for each factor that can be quickly re-applied. Problems for tBurton are formulated in terms of Timed Concurrent Automata, a variation on timed-automata theory. In this paper we briefly describe the formalism, the algorithm, and the translation necessary to apply tBurton to problems expressed in PDDL.

Introduction

tBurton is a model-based planner designed as a progression of the reactive model-based planner, Burton, to handle problems with time. As such, tBurton is capable of planning for systems that have indirect-effects (as a result of interactions between machines), irreversible actions, timed, possibly periodic behavior, and imprecise execution times. The plan tBurton produces is temporally least-commitment: providing time-bounds within which imprecise execution times can be tolerated. Furthermore, the user can specify goals with deadlines and time-windows (time-evolved goals) in order to constrain the space of possible plans considered. These problem features are natively expressed in tBurton using Timed Concurrent Automata (TCA).

In terms of PDDL, tBurton is capable of solving PDDL problems through the use of the translator from PDDL to TCA (subsection). with timed initial literals, durative-actions with duration inequalities, but not numeric fluents, . Additional PDDL features such as state-trajectory-constraints are theoretically supportable, but not currently implemented.

The overarching approach used in tBurton is Divide and Conquer: We use causal-graph analysis to 'divide' the problem into factors suitable to be 'conquered' by an off-the-shelf sequential heuristic search planner. The resulting plans for each factor are then unified in a conflict directed search, which also learns re-usable plan fragments.

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

While the importance of causal graphs in planning has been established (Williams and Pandurang Nayak 1997; Chen and Giménez 2010; Brafman and Domshlak 2006; Helmert 2004), this conflict-directed unification of causal graphs and heuristic forward search is new.

In this paper we review the tBurton planner. We begin by presenting the TCA formalism and then sketch the algorithm. Finally, we close by providing the translation from PDDL to TCA.

Problem Formulation

tBurton operates over two formalisms: Timed Concurrent Automata (TCA), which expresses the behavior of the system, encoded as a set of automata with guarded transitions; and State Plans (SP) which uses a temporal network to express when automata states need to be achieved.

Given a TCA and a partial state plan SP_{part} (which expresses only the initial state and goals), tBurton's objective is to elaborate SP_{part} by adding transitions until all open goals are closed and the plan is consistent with the TCA. We refer to the resulting plan as the Total State Plan, SP_{total}.

Formally, a TCA consists of a set of automata, A, whose operation is defined by the interaction of three different classes variables: location variables, control variables, and clock variables.

Definition 1. An automaton, A is the 5-tuple $\langle l, C, u, \mathbb{T}, \mathbb{I} \rangle$.

- l is a location variable, whose finite-domain represent the locations over which this automata transitions.
- $\mathcal{C}$ is set of clock variables uniquely used by this automaton. A clock variable, c, is a positive, real-valued variable that acts as a stop-watch to track time.
- u is the unique control variable for this automaton. The control variable has a finite-domain of values, that represent control values which can be selected externally to the automaton to affect its transitions.
- $\mathbb{T}$ is a transition function, that associates with a start and end location l_s, l_e a guard g and a subset of C to be reset to 0. A guard is expressed in terms of propositional formulas with equality, φ, where: $\varphi ::= \texttt{true} \mid \texttt{false} \mid (l_o = v) \mid (u = v) \mid (c\ op\ r) \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$. The guard can be expressed only in terms of location variables not belonging to this automaton, $l_o \in \mathcal{L} \backslash l$, and the control

and clock variables of this automaton. The automaton is said to instantaneously transition from l_s to l_e and reset its clock variables when the guard is first evaluated true.

- $\mathbb{I}$ is a function that associates with each location an invariant, a clock comparison of the form $c < r$ or $c \leq r$ that bounds the maximum amount of time an automata can stay in that location.

An automaton is *well formed* if there exists a next-state for all possible combination of assignments to location, control, and clock variables. With regards to the transitions, an automaton is said to be *deterministic* if for any l_s only the guard g of one transition can be true at any time. tBurton can only reason over TCA models consisting of well-formed, deterministic automata.

Definition 2. A State Plan SP consists of a set of episodes $ep = \langle e_i, e_j, lb, ub, sc \rangle$, which express for a pair of timepoints, e_i and e_j, the temporal separation allowed between them $lb \leq e_j - e_i \leq ub$, and a state constraint sc, (expressed in propositional state logic over location, clock, and control variables) that must hold over that interval.

We further categorize the episodes of the State Plan into Goal Histories, Value Histories, and Justifications. An episode participating in a Goal History is called a goal-episode. It's state constraint expresses a desired state. A value-episode expresses a state that should be achieved.

A justification-episode expresses a temporal relation between goal and value episodes. An important function of justification episodes is to indicate that a value-episode has *closed* a goal-episode. Informally, a value episode closes a goal episode, if there is a justification relating the start events of each episode, a justification relating the end events of each episode, and the state constraint on the value episode entails that of the goal.

tBurton Planner

tBurton depends on many component algorithms, but for the purposes of this presentation we focus on the two most significant in understanding its operation. The first is causal-graph decomposition and how we use it to induce a search ordering. The second is the high-level search, for which we will spend the bulk of this section describing.

Causal Search Ordering

Given a TCA, tBurton first builds an acyclic causal graph which orders the automata based on their interdependencies. The graph imposes a 'causal' search ordering, in which tBurton will first plan for goals involving the children before its parents.

The construction is straight-forward. Each $A \in TCA$ becomes an graph embedded in a vertex of the causal graph. If automaton A_1 has a guard that uses the location variable of automaton A_2, we add a directed edge from the vertex containing A_2 to the vertex containing A_1. To make the causal graph acyclic, any automata involved in cycles are replaced by their automata product. Finally, the graph is walked in a depth-first manner, numbering children before parents. This numbering can be used later to ensure goals on children automata are closed before their parents.

Search as Regression through Histories

tBurton's high-level search algorithm inherits ideas from Burton (Williams and Pandurang Nayak 1997), UCPOP (Penberthy and Weld 1992), and conflict-directed search (Chen and Van Beek 2011).

As a partial order planner, tBurton searches over plan space. This space is defined by permutations of SP_{part}. To move in this space, tBurton can perform three different plan-space actions, or choices that modify SP_{part}.:

1. **Select a goal ordering within each Automaton.** When tBurton starts planning, it has a set of goal-episodes to close. While the causal graph tells us we should plan for one automaton before another, it does not tell us how to order a subset of those goals if they occur over a single automaton. Since actions are not reversible and reachability checking is hard, the order in which goals are achieved matters. tBurton must impose a total ordering on the goals involving the location of a single automaton. Recall that since an automaton can have no concurrent transitions, a total order does not restrict the space of possible plans for any automaton.

   Relative to SP_{part}, imposing a total order involves adding episodes to the goal history of the form $ep = \langle e_i, e_j, 0, \infty, true \rangle$, for events e_i and e_j that must be ordered.

2. **Select a value to close a goal.** Since goals can have constraints expressed as propositional state-logic, it is possible we may need to achieve disjunctive subgoals. In this case, tBurton must select a value that entails the goal.

   To properly close the goal, tBurton must also represent this value selection as an episode added to the value history of the appropriate automata or control variable, and introduce two justifications which honor the bounding constraints.

3. **Select a sub-plan to achieve a value.** The sub-plan tBurton must select need only consider the transitions in a single automaton, A. Therefore, the sub-plan must be selected based on two sequential episodes, ep_s ep_g, in the value history of A (which will be the initial state and goal for the sub-plan), and the set-bound temporal constraint that separates them. The method tBurton uses to select this sub-plan can be any blackbox, but we will use a heuristic forward search, temporal planner. To properly add this sub-plan to SP_{part}, tBurton must add the plan to the value history and introduce any new goals this plan requires of parent automata.

These three choices form the basis of the high level search. But, there are two additional details that have a significant impact on tBurton's performance.

In order to maintain search state, the algorithm uses a queue to keep track of the partial plans, SP_{part}, that it needs to explore. For simplicity, one can assume this queue is FIFO, although in practice, a heuristic could be used to sort the queue. We make two additional modifications to the SP_{part} we store on the queue, to make search more efficient. First, we annotate SP_{part} with which of the three choices it needs to make next. The second addition involves the use

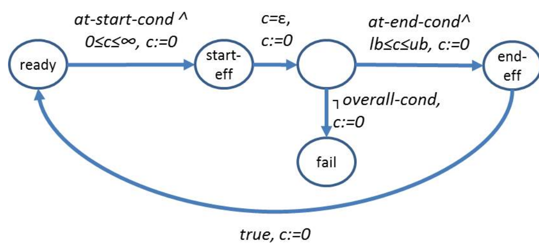


Figure 1: An example TCA automaton for a generic PDDL grounded action.

of an Incremental Temporal Order (ITO) algorithm. When tBurton needs to select a goal ordering for a given partial plan, it could populate the queue with partial plans representing all the variations on goal ordering. Since storing all of these partial plans would be very memory intensive, we add to the partial plan a data structure from ITO, which allows us to store one partial plan, and poll it for the next temporally consistent variation in total goal ordering.

As a consequence of using plans to close goals instead of actions, we can take advantage of memorizing and conflict learning. In particular, a sequential plan for a timed automaton is defined by its initial location, goal, and bounds on the temporal duration between the two. We can associate with these pieces of information, a plan (or lack of a plan) that can satisfy it. In the future when a similar situation arises, we can lookup these cached results faster than generating another sequential plan.

PDDL to TCA

In order to run tBurton on PDDL problems, we developed a PDDL (without fluents) to TCA translator. Here, we provide only a sketch of this translator.

In order to maintain as much concurrency as possible in the domain, the translator first uses temporal invariant synthesis (Bernardini and Smith 2008) to compute a set of invariants. An instance of an invariant identifies a subset of ground predicates in which only one can be true at the same time. We select a subset of these invariant instances that provide a covering of the state-space, and encode each invariant instance into an automaton. Each possible grounding of the invariant instance becomes a location in the automata.

Each ground durative action is also translated into an automaton (Figure 1). Three of the transitions are guarded by conditions from the corresponding PDDL action, translated into propositional state logic over location variables. Another transition uses ϵ to denote a small-amount of time to pass for the start-effects of the action to take effect prior to checking for the invariant condition. A fifth transition is used to reset to the action.

Finally, the transitions of each invariant-instance based automata is labeled with a disjunction of the states of the ground-action automata that effects its transition.

Discussion

In order to complete the process of applying tBurton to PDDL, there are two additional pieces necessary: An automaton to PDDL translator is needed to interface with the heuristic forward search planner. This can be derived by translating each TCA transition to a simple durative action, while exploiting the reverse of the mapping described in the preceding subsection where possible. There also needs to be a scheduling algorithm to choose specific times for each action from the least-commitment plan (Dechter, Meiri, and Pearl 1991). Since dynamic scheduling is not necessary any APSP algorithm is sufficient. Note that all of these decorations are necessary because tBurton was designed to solve problems with an emphasis on a different set of features. While we can apply tBurton to PDDL problems, it is still uncommon for PDDL problems to have time-evolved goals and actions with duration inequalities, the features for which tBurton was really designed to solve.

tBurton's use of the causal graph to factor the problem provides many indirect benefits: The ability to use sequential planners to plan for each factor, (for which the fastest seem to be heuristic forward search planners); The causal, child-to-parent search order; And the ability to learn and apply concise plan fragments. In practice, we find that these approaches greatly reduce the time required for plan-space search, and makes tBurton not only an effective TCA planner, but an effective PDDL planner as well.

References

Bernardini, S., and Smith, D. 2008. Translating pddl2. 2. into a constraint-based variable/value language. In *Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling, 18th International Conference on Automated Planning and Scheduling (ICAPS08)*.

Brafman, R. I., and Domshlak, C. 2006. Factored planning: How, when, and when not. In *AAAI*, volume 6, 809–814.

Chen, H., and Giménez, O. 2010. Causal graphs and structurally restricted planning. *Journal of Computer and System Sciences* 76(7):579–592.

Chen, X., and Van Beek, P. 2011. Conflict-directed backjumping revisited. *arXiv preprint arXiv:1106.0254*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial intelligence* 49(1-3):61–95.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. ICAPS.

Penberthy, J., and Weld, D. 1992. Ucpop: A sound, complete, partial order planner for adl. In *proceedings of the third international conference on knowledge representation and reasoning*, 103–114. Citeseer.

Williams, B., and Pandurang Nayak, P. 1997. A reactive planner for a model-based executive. In *International Joint Conference on Artificial Intelligence*, volume 15, 1178–1185. LAWRENCE ERLBAUM ASSOCIATES LTD.

Preferring Preferred Operators in Temporal Fast Downward

Patrick Eyerich and **Thomas Keller** and **Johannes Aldinger** and **Christian Dornhege**
University of Freiburg, Germany
{eyerich,tkeller,aldinger,dornhege}@informatik.uni-freiburg.de

Abstract

Temporal Fast Downward (TFD) is a temporal planning system that is capable of dealing with numerical values. In this paper, we briefly describe the main improvement of the version of TFD that participates at IPC-14 over the version from IPC-08: the incorporation of new methods of deciding whether certain operators should be preferred during search.

Temporal Fast Downward

Temporal Fast Downward (TFD) (Eyerich, Mattmüller, and Röger 2009) is a domain-independent progression search planning system built on top of the classical planner *Fast Downward* (Helmert 2006). It extends the original system by supporting durative actions as well as numeric and object fluents. The main improvement of the version of TFD that participates at IPC-14 over the original version is the incorporation of new methods of deciding whether certain operators should be preferred during search (Eyerich 2012). In this paper, we briefly describe these methods.

In the following, we use the definition of Eyerich et. al. of a *temporal SAS$^+$ planning task* (Eyerich, Mattmüller, and Röger 2009), a tuple $\Pi = \langle \mathcal{V}, s_0, s_\star, \mathcal{A}, \mathcal{O} \rangle$, where $\mathcal{V}$ is a set of *state variables*. The initial state s_0 is given by a variable assignment (a *state*) over all fluents in $\mathcal{V}$ and the set of goal states $s_\star$ is defined by a partial state (a state restricted to a subset of fluents) over $\mathcal{V}$. Analogously to the Boolean setting, we identify such variable mappings with the *set of atoms* $v = w$ that they make true. For an atom x we write $\mathrm{var}(x)$ to denote the variable associated with x. $\mathcal{A}$ is a finite set of *axioms* and $\mathcal{O}$ is a finite set of *durative actions*.

A *time-stamped state* $\mathcal{S} = \langle t, s, E, C_\leftrightarrow, C_\dashv \rangle$ consists of a *time stamp* $t \geq 0$, a *state* s, a set E of *scheduled effects*, and two sets $C_\leftrightarrow$ and $C_\dashv$ of persistent and end conditions.

A durative action is *applicable* in a time-stamped state $\mathcal{S}$ if it can be integrated into $\mathcal{S}$ in a consistent way (Eyerich, Mattmüller, and Röger 2009). The successors of a time-stamped state are generated by either inserting an applicable durative action at the current time point or by increasing the time-stamp to the earliest time point where a scheduled action ends.

Copyright © 2012, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Context-enhanced additive heuristic

For guiding the search, TFD uses a variant of the (inadmissible) context-enhanced additive heuristic (h^{cea}) (Helmert and Geffner 2008) extended to cope with numeric variables and durative actions. To make h^{cea} useful for temporal planning, Eyerich et. al. show how to transform durative actions to several types of so-called *instant actions* (Helmert and Geffner 2008), which we assume to be given in this paper. Instant actions are sets of effects of the form $v{=}w', z \rightarrow v{=}w$, where v is a variable, z is a partial state not mentioning v, and w and w' are values for v. Such an effect means that if the current state s satisfies z and maps v to w', then the successor state s', resulting from the application of the operator, maps v to w (while all mappings that are not changed by any effect of the operator stay the same). We also write $a : v{=}w', z \rightarrow v{=}w$ to make clear that the rule is an effect of the instant action a.

Given a state s and an atom $v{=}w$, we denote with $s[v{=}w]$ the state that is like s except for variable v, which it maps to w. Similarly, we write $s[s']$ where s' is a partial state to denote the state that is like s' for variables defined in s' and like s for all other variables.

For a time-stamped state s and a goal specification $s_\star$, the cost-sensitive variant of h^{cea} is defined as

$$h^{cea}(s) \stackrel{\text{def}}{=} \sum_{x \in s_\star} h^{cea}(x|x_s),$$

where x_s is the atom that refers to $\mathrm{var}(x)$ in state s and $h^{cea}(x|x_s)$ estimates the costs of changing the value of $\mathrm{var}(x)$ from the value it has in s to the one required in $s_\star$.

The context-enhanced additive heuristic makes the underlying assumption that for any atom x conditions referring to $\mathrm{var}(x)$ are achieved first, while all other conditions are evaluated in the resulting state s'', leading to the following definition:

$$h^{cea}(x|x') \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x = x' \\ \min\limits_{o:x'',z \rightarrow x} \big(c(o, s'') + h^{cea}(x''|x') + \sum\limits_{x_i \in z} h^{cea}(x_i|x_i'')\big) & \text{else} \end{cases}$$

where $c(o, s)$ is the cost of applying operator o in state s. The state s'' is the state after reaching x'' from x'. Note that

with the minimum of the empty set being infinity, $h^{cea}(x|x')$ might also be infinity and if it is, there is no plan that satisfies the goal in the original task.

In this definition, the first case is trivial. In the second case, the first summand, $c(o, s'')$, captures the cost of applying the minimizing operator o in state s'', the second one estimates the cost of achieving x'' from x', and the third one the cost of making all other conditions z of the rule true. In this third term, atom x''_i is the atom associated with $\mathrm{var}(x_i)$ in the state that results from achieving x'' from x'.

To reschedule solutions in order to reduce their makespan, the TFD version used for this paper features a partial-order lifting procedure that is inspired by the work of Do and Kambhampati and Coles et. al. (Do and Kambhampati 2003; Coles et al. 2009) and extended to be able to deal with conditional effects.

Preferred Operators

Conceptually, the idea of preferred operators is to transfer information about which operator's application seems to be promising from the heuristic abstraction to the actual search. This concept was first realized by McDermott by determining *favored actions* in the context of greedy regression graphs as those applicable actions that are part of the minimal cost subgraph achieving the goals (McDermott 1996; 1999). Hoffmann and Nebel defined *helpful actions* in their FF planner as those actions that achieve a fact required by an action in the relaxed plan that appears in the first layer of the planning graph (Hoffmann and Nebel 2001). FF considers only helpful actions in its first attempt of finding a solution and switches to a complete greedy best-first search if it fails. Another approach is used in Fast Downward, where besides the usual open list containing all applicable operators there is a separate open list containing only preferred operators. Different strategies of how to best combine these two open list have been investigated (Richter and Helmert 2009; Röger and Helmert 2010).

Using the definition of the context-enhanced additive heuristic, the set $\mathcal{P}(s)$ of *preferred operators* is defined as

$$\mathcal{P}(s) \stackrel{\text{def}}{=} \bigcup_{x \in s_\star} \mathcal{P}(x|x_s),$$

where

$$\mathcal{P}(x|x') \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } x = x' \text{ or } h^{cea}(x|x') = \infty \\ \{o\} & \text{if } \exists\, o : x', w \to x : \\ & \qquad h^{cea}(x|x') = c(o, s') \\ \bigcup_{x_i \in w} \mathcal{P}(x_i|x'_i) & \text{if } \exists\, o : x', w \to x : \\ & \qquad h^{cea}(x|x') = \Big(c(o, s') + \\ & \qquad\qquad \sum_{x_i \in w} h^{cea}(x_i|x'_i)\Big) \\ \mathcal{P}(x''|x') & \text{if } \exists\, x'' : h^{cea}(x''|x') \\ & \qquad + h^{cea}(x|x'') = h^{cea}(x|x') \end{cases}$$

Each condition additionally requires the previous conditions to be unsatisfied. We furthermore assume that no action with zero cost exists and that, if the existentially quantified conditions are satisfied for more than one operator or atom, an arbitrary one is chosen according to a fixed tie-breaking strategy.

The first case is trivial. The second case defines an operator o that transforms x' to x and where all preconditions are satisfied as preferred. In the third case, some of the operator's preconditions are not satisfied. In that case, preferred operators are recursively defined over those preconditions. In the last case, x' cannot be changed to x by a single operator but only via an intermediate state, so preferred operators are recursively defined over this state.

In its default configuration, TFD uses a straight-forward adaptation of the boosted dual queue approach for preferred operators of Fast Downward (Helmert 2006). Preferred operators work best in the context of deferred evaluation. However, there are certain domain characteristics for which that is not the case. Especially in problems where goals are conflicting, requiring mutex operators, the preferred operator handling of TFD does not yield good results in the context of deferred evaluation.

The main reason for this poor behavior is that h^{cea} computes costs of subgoals independently from each other. In that way, a set of preferred operators might contain mutex operators each leading to a successor state with the same heuristic estimate (due to deferred evaluation) while the successors of each successor have a higher heuristic estimate. To see this, think of a problem in an elevators domain where we have two goals g_1 and g_2 to transport two passengers p_1 and p_2 from their common starting location f_5 to their desired floors f_1 and f_{10}, respectively. When investigating the subproblems independently from each other, as h^{cea} does, it might be meaningful to use the same elevator e_1, located at f_5, to transport both p_1 and p_2. In such a situation, both the operators move-down(e_1, f_5, f_1), leading to state s_1, and move-up(e_1, f_5, f_{10}), leading to state s_2, are preferred, and since we use deferred evaluation, s_1 and s_2 share the same heuristic estimation. When s_1 is expanded, however, e_1 has started to move to f_1 in order to satisfy g_1, and the heuristic realizes that g_2 becomes more expensive (potentially to a higher degree than the amount that g_1 becomes cheaper), leading to a worse overall state evaluation for all successors of s_1. Expansion of s_2 is analogous. In such a situation a potentially very large set of states has to be visited before the search actually progresses in the right direction.

Narrowing strategies

Our new selection strategies are basically methods to intelligently narrow the set of preferred operators, motivated by examples like the one above: If by using only preferred operators a planning task is rendered incomplete anyway, and if generating preferred operators for all subgoals at once can lead to situations where the search gets stuck, why not limit ourselves to generating preferred operators for only up to n subgoals? Of course, the obvious questions are *which* and *how many* operators out of a set of preferred ones we should choose. We have found three narrowing strategies to be use-

ful in practice: To utilize only the preferred operators that correspond to the first n yet unsatisfied goals, called $\mathcal{O}$, or to choose the preferred operators corresponding to the n goals that are cheapest or most expensive to satisfy according to the heuristic, called $\mathcal{C}$ and $\mathcal{E}$, respectively. More concretely, a *narrowing strategy* $\mathcal{X}^n(s)$ is defined as

$$\mathcal{X}^n(s) \stackrel{\text{def}}{=} \bigcup_{x \in X \subseteq s_\star} \mathcal{P}(x|x_s)$$

with an appropriate X of cardinality n chosen according to the selection strategy of $\mathcal{X}$. For $\mathcal{O}$, this strategy is defined such that $x \leq_{\mathcal{O}} y$ for all $x \in X, y \in (s_\star \setminus X)$ holds for some appropriate ordering relation $\leq_{\mathcal{O}}$. For $\mathcal{C}$ it has to hold that $h^{cea}(x|x_s) \leq h^{cea}(y|y_s)$ for all $x \in X, y \in (s_\star \setminus X)$, and for $\mathcal{E}$ it has hold that $h^{cea}(x|x_s) \geq h^{cea}(y|y_s)$ for all $x \in X, y \in (s_\star \setminus X)$.

Basically, all narrowing strategies examine the current state s and choose up to n goals x_i from $s_\star$ to compute preferred operators for: $\mathcal{O}$ determines the first n unsatisfied goals (according to an appropriate ordering relation $\leq_{\mathcal{O}}$), while $\mathcal{C}$ and $\mathcal{E}$ determine the heuristic cost of each subgoal as if it would be the only goal to satisfy (as said, this is done by h^{cea} anyway) and choose the n that are cheapest and most expensive, respectively. Note that with small n the search is driven to satisfy the goals more sequentially, however, each goal might be satisfied by parallel action applications.

Finding a good ordering relation for $\mathcal{O}$ is very much related to the more general task of detecting goal orderings (Köhler and Hoffmann 2000). In this paper, we only use the natural ordering that is defined by the order in which variables occur in the problem description for that purpose and defer the interesting question of how to combine our technique with goal ordering detection techniques to future work.

Algorithm 1: Best-first search with restarts, deferred evaluation, and several open lists in a priority based multi-queue approach.

```
1  activeList = chooseOpenListToStartWith()
2  forall open in openLists do
3      open.priority = 0
4  activeList.priority += V
5  activeList.add(s_0)
6  closedList ← ∅
7  lastProgressAtStep = 0, currentStep = 0
8  while activeList is not empty do
9      currentStep += 1
10     if (currentStep - lastProgressAtStep) > K then
11         activeList = nextOpenListToBoost()
12         restartAtLine2() or switchToRoundRobinMode()
13     s ← activeList.pop()
14     activeList.priority -= 1
15     if s ∉ closedList then
16         closedList.add(s)
17         if s ⊨ G then
18             return s as solution
19         h = s.compute_heuristic()
20         f = s.timestamp + h
21         if makes_progress(s) then
22             activeList.priority += V
23             lastProgressAtStep = currentStep
24         forall child states s' of s do
25             forall open in openLists do
26                 if open.matches(s') then
27                     open.add(s', f)
28     activeList = selectList()
29 return no solution found
```

Priority based multi-queue search with restarts

As we will show in the experiments section, utilizing our new techniques in TFD pays off in terms of coverage. Unfortunately, the produced solutions are typically of a lower quality than those of the original definition as the search is driven to satisfy goals more sequentially. Additionally, it can be observed that the different narrowing strategies have strengths in different domains. Motivated from these two facts, we have developed an algorithm that incorporates several narrowing strategies into a best-first search framework that uses an own open list for each strategy, as outlined in Algorithm 1.

The algorithm is based on the boosted dual-queue best-first search approach of Fast Downward (Helmert 2006). It maintains a set of open lists, each associated with a corresponding selection method. It has been shown that alternating between different open lists is a good idea if the open lists contain operators ordered by different heuristics (Röger and Helmert 2010). In our context, however, alternating did not work well, so we have chosen a priority based approach where each open list is associated with a priority and at each search step the algorithm selects the non-empty list with the highest priority (line 28). The search keeps track of the number of steps that were performed since the last time progress has been made (progress is made if a state is extracted from a list that has a lower heuristic estimate than each other state that has previously been taken from that list). If more than K steps have been made without making progress, the search restarts (lines 10–12), boosting a different open list each time by giving it a high initial priority while all other lists start with priority zero. If the search has restarted with each open list being boosted initially once, it switches to a round robin selection mode (line 12, details have been omitted from the pseudocode to ensure readability). During successor generation, nodes are inserted into the appropriate open lists according to their associated selection strategies (lines 24–27). Note that using a regular open list containing all applicable successors (which is done in our implementation) ensures completeness of the algorithm on temporally simple problems.

For the two parameters of the algorithm we have found $K = 3000$ and $V = 1000$ to work well in practice (these parameters, however, are quite robust and we got reasonable results for a wide range of values for both K and V). Note that the algorithm can be called from outside in an anytime fashion where the makespan of previously found solutions can be used to prune the search space.

IPC 2011	$\mathcal{C}^1$	$\mathcal{C}^2$	$\mathcal{C}^3$	$\mathcal{E}^1$	$\mathcal{E}^2$	$\mathcal{E}^3$	$\mathcal{O}^1$	$\mathcal{O}^2$	$\mathcal{O}^3$
CREWPLANNING	14.2	15.5	**15.6**	0.0	2.4	1.6	0.0	2.4	4.2
	19	**20**	**20**	0	3	2	0	3	5
ELEVATORS	**15.1**	10.5	7.5	0.0	0.0	0.0	13.4	7.0	4.5
	18	12	8	0	0	0	**18**	10	6
FLOORTILE	4.3	4.9	4.7	4.4	**5.2**	4.5	4.8	4.8	4.0
	5	5	5	5	**6**	5	5	5	4
MATCHCELLAR	**15.6**	**15.6**	**15.6**	**15.6**	**15.6**	**15.6**	**15.6**	**15.6**	**15.6**
	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
OPENSTACKS	3.6	5.0	6.4	14.2	14.4	**15.3**	4.0	6.1	7.8
	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
PARCPRINTER	1.0	0.0	0.0	0.0	0.0	0.0	**9.4**	2.7	1.7
	1	0	0	0	0	0	**10**	3	2
PARKING	14.0	**14.9**	11.4	8.9	8.6	9.0	6.7	8.5	7.5
	17	**19**	14	12	12	12	9	11	10
PEGSOL	17.7	17.4	18.5	18.6	18.8	18.8	18.4	19.0	**19.3**
	18	18	19	19	19	19	19	**20**	**20**
SOKOBAN	3.8	**3.9**	2.9	2.9	2.9	2.9	3.8	**3.9**	2.9
	4	**4**	3	3	3	3	**4**	**4**	3
STORAGE	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0	0	0	0	0	0	0	0	0
TMS	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	0	0	0	0	0	0	0	0	0
TURNANDOPEN	10.1	10.0	10.8	8.8	8.6	8.6	**11.0**	8.4	8.6
	20	**20**	**20**	12	12	12	19	14	14
Overall	**99.5**	97.7	93.3	73.4	76.5	76.2	87.2	78.5	76.1
	142	138	129	91	95	93	124	110	104

Table 1: Performance of new selection strategies on IPC 2011 benchmarks. Gray rows show IPC scores, white rows coverage.

Experiments

In our first experiment we simply replace the basic preferred operator strategy of TFD by one of the three presented narrowing methods. Results for $\mathcal{C}^n$, $\mathcal{E}^n$, and $\mathcal{O}^n$ with $1 \leq n \leq 3$ are shown in Table 1.

It can be seen that $\mathcal{C}$ and $\mathcal{O}$ yield very promising results with a higher coverage compared to the original method, especially in ELEVATORS and PARCPRINTER. The reason for the good performance in ELEVATORS seems to be that by narrowing the set of preferred operators the weakness of the heuristic to switch between subgoals during search can be overcome by focusing on a specific goal. In doing so, it is better to focus on the cheapest goal ($\mathcal{C}$) than on an arbitrary one ($\mathcal{O}$). It is useless, however, to focus on the most expensive goal ($\mathcal{E}$), as this changes to often during search. In PARCPRINTER both the cheapest and the most expensive goal vary a lot during search, so it is best to focus on a fixed goal like $\mathcal{O}$ does. Unfortunately, $\mathcal{O}$ does yield very bad results in CREWPLANNING, where a specific goal ordering needs to be respected that $\mathcal{O}$ is not aware of. Here, techniques to detect goal orderings (Köhler and Hoffmann 2000) might be very helpful. While coverage can be increased using our new techniques, their produced solutions are typically of lower quality than those of the original method as they drive the search to satisfy goals more sequentially. This fact becomes apparent especially in OPENSTACKS, a domain for which it is very easy to find a solution but the range of quality is

IPC 2011	$\mathcal{PC}^1\mathcal{O}^1$	$\mathcal{PC}^1\mathcal{E}^1$	$\mathcal{PO}^1\mathcal{E}^1$	**TFD$^+$**
CREWPLANNING	**19.9** (**20**)	**19.9** (**20**)	**19.9** (**20**)	**19.9** (**20**)
ELEVATORS	13.4 (**18**)	**15.4** (**18**)	13.4 (**18**)	13.4 (**18**)
FLOORTILE	**4.8** (**5**)	4.6 (**5**)	4.7 (**5**)	**4.8** (**5**)
MATCHCELLAR	15.6 (**20**)	15.6 (**20**)	15.6 (**20**)	**15.6** (**20**)
OPENSTACKS	17.9 (**20**)	18.2 (**20**)	**18.3** (**20**)	17.9 (**20**)
PARCPRINTER	9.5 (**10**)	0.9 (1)	9.5 (**10**)	**9.5** (**10**)
PARKING	14.1 (18)	13.8 (17)	12.0 (16)	**14.6** (**19**)
PEGSOL	**18.6** (**19**)	18.5 (**19**)	18.3 (**19**)	**18.6** (**19**)
SOKOBAN	2.9 (**3**)	**3.0** (**3**)	2.9 (**3**)	2.9 (**3**)
STORAGE	0.0 (0)	0.0 (0)	0.0 (0)	0.0 (0)
TMS	0.0 (0)	0.0 (0)	0.0 (0)	0.0 (0)
TURNANDOPEN	14.0 (**20**)	13.2 (19)	**14.1** (**20**)	14.0 (**20**)
Overall	130.7 (153)	123.1 (142)	128.7 (151)	**131.2** (**154**)

Table 2: IPC scores and coverage (in parentheses) of combining several narrowing strategy via restarting as described in Algorithm 1. TFD$^+$ is an abbreviation for $\mathcal{PC}^1\mathcal{O}^1\mathcal{E}^1$.

IPC 2011	CPT4	LMTD	YAHSP2	YAHSP2-mt	POPF2	DAE-YAHSP	TFD	TFD$^+$
CREWPLANNING	7.0	0.0	16.0	15.9	**20.0**	**20.0**	19.9	19.9
	7	0	**20**	**20**	**20**	**20**	**20**	**20**
ELEVATORS	0.0	6.7	8.6	8.9	2.2	12.3	1.0	**13.4**
	0	9	**20**	**20**	3	15	1	18
FLOORTILE	**12.1**	4.8	6.9	8.3	0.0	7.3	4.9	4.8
	15	5	13	**15**	0	12	5	5
MATCHCELLAR	0.0	12.5	0.0	0.0	15.3	0.0	**15.6**	**15.6**
	0	15	0	0	**20**	0	**20**	**20**
OPENSTACKS	0.0	0.0	12.6	12.1	15.0	**19.9**	17.7	17.9
	0	0	**20**	19	**20**	**20**	**20**	**20**
PARCPRINTER	2.0	0.0	3.7	4.7	0.0	2.0	0.0	**9.5**
	5	0	7	8	0	4	0	**10**
PARKING	0.0	0.0	11.0	12.7	14.7	**15.9**	12.2	14.6
	0	0	**20**	**20**	**20**	**20**	16	19
PEGSOL	19.0	19.9	17.2	18.0	18.6	**20.0**	17.9	18.6
	19	**20**	**20**	**20**	19	**20**	18	19
SOKOBAN	0.0	0.0	10.9	**11.6**	2.5	4.5	2.9	2.9
	0	0	**12**	**12**	3	6	3	3
STORAGE	0.0	0.0	2.7	7.2	0.0	**15.5**	0.0	0.0
	0	0	5	11	0	**19**	0	0
TMS	0.0	0.0	0.0	0.0	**5.0**	0.0	0.0	0.0
	0	0	0	0	**5**	0	0	0
TURNANDOPEN	0.0	7.0	0.0	0.0	7.8	0.0	13.3	**14.0**
	0	13	0	0	9	0	**20**	**20**
Overall	40.1	50.9	89.6	99.3	101.1	117.2	105.3	**131.2**
	46	62	137	145	119	136	123	**154**

Table 3: Gray rows show IPC scores, white rows coverage of participants of IPC 2011. The two rightmost columns show results of original TFD and TFD enhanced the techniques presented in this paper (TFD$^+$).

very high and it is important to start the right actions first in order to create concurrent solutions. Interestingly, $\mathcal{E}$ works quite well in this domain, as the actions that are needed to be started first in order to create a compact solution are also the most expensive ones.

Another interesting observation is that in the good performing methods $\mathcal{C}$ and $\mathcal{O}$ it is advantageous to concentrate on a smaller set of subgoals, while the converse holds for the poor performing method $\mathcal{E}$. This is due to the fact that with increasing size of the preferred operators set the original set is resembled more and more.

The most important observation that can be made from this experiment has motivated the design of the search procedure presented in the previous section: Different selection strategies have strengths in different domains and it appears to be very desirable to combine these strengths in a general way. Table 2 shows results of an implementation of Algorithm 1 combining several narrowing strategies, with $\mathcal{P}\mathcal{O}^1\mathcal{C}^1\mathcal{E}^1$, abbreviated as TFD$^+$ in the table, achieving both the highest coverage and IPC score.

To see how these improvements affect the performance of TFD relatively to other temporal planning systems, we compared both the original TFD and TFD$^+$, a version of TFD that implements Algorithm 1 with queues for $\mathcal{P}$, $\mathcal{O}^1$, $\mathcal{C}^1$, and $\mathcal{E}^1$ to the participants of the temporal satisficing track of IPC 2011 that achieved at least one point in the competition. For this experiment, we did not re-run the other planning systems, but use the raw results of the competition directly.[1] Table 3 presents IPC scores (gray rows) and coverage (white rows). It can be seen that TFD$^+$ clearly outperforms all competitors both in terms of coverage and IPC score.

Note that for some planners the scores presented in this paper vary from the scores they received in the competition as we did find better plans for many problems and used them as reference plans to compute all scores. For example, CPT4, which is optimal for the conservative semantics of Smith and Weld (Smith and Weld 1999), produced some non-optimal plans in Floortile and Parcprinter. This was not recognized as its plans were the best of those generated during the competition.

In another experiment, presented in Table 4, we focus on quality by comparing TFD featuring our techniques, called TFD$^+$, pairwise to all other planners of IPC 2011, only considering problems where both planners have found a solution by computing the ratio between the makespan of those solutions. Scores greater than 1.0 therefore indicate that we found plans of higher quality. It can be seen that our plans offer the highest quality throughout all domains.

Finally, in our last experiment we show that the good performance of our techniques is not only a phenomenon on a specific benchmark set, but occurs on a wider range of domains. Therefore, we use the benchmark suites of IPCs 2006 and 2008 (excluding Pathways and TPP, where not

[1] IPC 2011 has been run on INTEL Xeon 2.93 GHz Quad Core processors with a memory limit of 6 GB and a timeout of 30 minutes. Note that TFD (like most processes) generally runs faster on such a system than on the system we used to generate our results, so the comparison is in favor of the planning systems that participated in the competition.

IPC 2011	CPT4	LMTD	YAHSP2	YAHSP2-mt	POPF2	DAE-YAHSP	TFD
CREWPLANNING	$_{7}$**1.00**	–	$_{20}$**1.29**	$_{20}$**1.29**	$_{20}$0.99	$_{20}$0.99	$_{20}$**1.00**
ELEVATORS	–	$_{9}$0.94	$_{18}$**2.08**	$_{18}$**1.98**	$_{3}$**1.01**	$_{15}$0.93	$_{1}$0.59
FLOORTILE	$_{5}$**1.67**	$_{2}$0.99	$_{4}$**2.38**	$_{5}$**2.22**	–	$_{4}$**2.36**	$_{5}$0.96
MATCHCELLAR	–	$_{15}$**1.06**	–	–	$_{20}$**1.25**	–	$_{20}$**1.24**
OPENSTACKS	–	–	$_{20}$**1.47**	$_{19}$**1.46**	$_{20}$**1.23**	$_{20}$0.92	$_{20}$**1.04**
PARCPRINTER	$_{2}$**1.76**	–	$_{7}$**1.88**	$_{7}$**1.88**	–	$_{4}$**1.95**	–
PARKING	–	–	$_{19}$**1.50**	$_{19}$**1.35**	$_{19}$**1.12**	$_{19}$**1.11**	$_{16}$**1.03**
PEGSOL	$_{18}$0.99	$_{19}$0.98	$_{19}$**1.16**	$_{19}$**1.11**	$_{18}$**1.00**	$_{19}$0.98	$_{18}$**1.00**
SOKOBAN	–	–	$_{3}$**1.02**	$_{3}$**1.03**	$_{2}$**1.17**	$_{3}$**1.10**	$_{3}$**1.00**
STORAGE	–	–	–	–	–	–	–
TMS	–	–	–	–	–	–	–
TURNANDOPEN	–	$_{13}$**1.38**	–	–	$_{9}$0.61	–	$_{20}$**1.03**
OVERALL	$_{32}$**1.15**	$_{58}$**1.08**	$_{110}$**1.54**	$_{110}$**1.48**	$_{111}$**1.08**	$_{104}$**1.08**	$_{123}$**1.05**

Table 4: Pairwise quality comparisons to a version of TFD that implements all techniques presented in this paper. Only instances that are solved by both approaches (the small number states their number) are considered. Scores greater than 1.0 indicate that TFD$^+$ generates plans of higher quality.

	TFD	TFD+	Quality
IPC 2006			
OPENSTACKS	17.5 (18)	**20.0** (**20**)	$_{18}$ **1.03**
PIPESWORLD	**17.7** (**18**)	15.1 (16)	$_{15}$ 0.96
ROVERS	11.9 (12)	**16.8** (**17**)	$_{12}$ 0.99
STORAGE	**16.7** (**17**)	**16.7** (**17**)	$_{17}$ **1.01**
TRUCKS	13.5 (14)	**29.4** (**30**)	$_{14}$ **1.00**
IPC 2008			
CREWPLANNING-*strips*	**29.9** (**30**)	**29.9** (**30**)	$_{30}$ **1.00**
ELEVATORS-*numeric*	16.7 (20)	**25.2** (**30**)	$_{20}$ **1.02**
ELEVATORS-*strips*	13.0 (16)	**20.8** (**30**)	$_{16}$ 0.96
MODELTRAIN-*numeric*	1.0 (1)	**5.3** (**7**)	$_{1}$ **1.00**
OPENSTACKS-*adl*	27.1 (**30**)	**27.6** (**30**)	$_{30}$ **1.02**
OPENSTACKS-*strips*	27.1 (**30**)	**28.1** (**30**)	$_{30}$ **1.04**
PARCPRINTER-*strips*	9.0 (13)	**22.4** (**23**)	$_{13}$ **1.77**
PEGSOL-*strips*	28.3 (29)	**29.3** (**30**)	$_{29}$ **1.01**
SOKOBAN-*strips*	**11.9** (**12**)	**11.9** (**12**)	$_{12}$ **1.00**
TRANSPORT-*numeric*	4.9 (6)	**11.0** (**18**)	$_{6}$ **1.05**
WOODWORKING-*numeric*	16.6 (28)	**21.6** (**30**)	$_{28}$ **1.36**
Overall	262.9 (294)	**331.1** (**370**)	$_{291}$ **1.08**

Table 5: The two columns in the middle show IPC scores and coverage (in parentheses) of regular TFD and TFD$^+$ on the benchmarks suites of IPC 2006 and 2008. TFD$^+$ features separate queues for $\mathcal{C}^1$, $\mathcal{O}^1$, and $\mathcal{E}^1$, as well as restarting according to Algorithm 1. The last column shows pairwise plan quality comparisons between TFD and TFD$^+$ on all instances that were solved by both approaches (the small number states their number). Scores greater than 1.0 indicate that TFD$^+$ generates plans of higher quality.

only makespan but a more complex metric needs to be optimized, a feature TFD cannot deal with yet). Results are presented in Table 5. Note that only for the benchmark set of 2008 reference plans are used. TFD^+ has a higher coverage in all domains but PIPESWORLD. (In PIPESWORLD, h^{cea} fails to mark certain relevant operators as preferred, requiring to extract those operators from the regular open list. While regular TFD also uses preferred operators, it expands nodes from the regular open list more often as the number of lists is smaller than in TFD^+.) In this experiment the title of this paper is reflected very well: Coverage is increased drastically (from 294 to 370) while the average plan quality is not only maintained, but even slightly improved.

Conclusion

In this paper we have presented novel methods to narrow sets of preferred operators. Embedding these methods in the search framework of TFD increases its coverage at the price of quality. This drawback, however, can be overcome by utilizing a restarting strategy that is incorporated into a priority-based multi-queue best-first search framework. We have implemented these techniques and have shown empirically that combining them increases the coverage of TFD by a huge amount and preserves the average quality of the produced plans.

Acknowledgments

This work was supported by the German Aerospace Center (DLR) as part of the Project "Kontiplan" (50 RA 1221).

References

Coles, A.; Fox, M.; Halsey, K.; Long, D.; and Smith, A. 2009. Managing Concurrency in Temporal Planning Using Planner-Scheduler Interaction. *Artificial Intelligence* 173(1):1–44.

Do, M. B., and Kambhampati, S. 2003. Improving Temporal Flexibility of Position Constrained Metric Temporal Plans. In Giunchiglia, E.; Muscettola, N.; and Nau, D., eds., *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling*, 42–51.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2009. Using the Context-Enhanced Additive Heuristic for Temporal and Numeric Planning. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, 130–137.

Eyerich, P. 2012. Preferring properly: Increasing coverage while maintaining quality in anytime temporal planning. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI).*

Helmert, M., and Geffner, H. 2008. Unifying the Causal Graph and Additive Heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling*, 140–147.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14:253–302.

Köhler, J., and Hoffmann, J. 2000. On Reasonable and Forced Goal Orderings and their Use in an Agenda-Driven Planning Algorithm. *Journal of Artificial Intelligence Research* 12:338–386.

McDermott, D. 1996. A Heuristic Estimator for Means-Ends Analysis in Planning. In Drabble, B., ed., *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems*, 142–149.

McDermott, D. 1999. Using Regression-Match Graphs to Control Search in Planning. *Artificial Intelligence* 109(1–2):111–159.

Richter, S., and Helmert, M. 2009. Preferred Operators and Deferred Evaluation in Satisficing Planning. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, 273–280.

Röger, G., and Helmert, M. 2010. The More, the Merrier: Combining Heuristic Estimators for Satisficing Planning. In Brafman, R. I.; Geffner, H.; Hoffmann, J.; and Kautz, H. A., eds., *Proceedings of the Twenty International Conference on Automated Planning and Scheduling*, 246–249.

Smith, D. E., and Weld, D. S. 1999. Temporal Planning with Mutual Exclusion Reasoning. In Dean, T., ed., *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 326–337. Morgan Kaufmann.