22nd International Conference on Automated Planning and Scheduling

June 25-29, 2012, Atibaia – Sao Paulo – Brazil

HSDIP 2012

Proceedings of the Workshop on Heuristics and Search for Domain-Independent Planning

Edited by
Patrik Haslum, Malte Helmert, Erez Karpas, Carlos Linares López, Gabriele Röger, Jordan Thayer, Rong Zhou

Organization

Patrik Haslum, Australian National University

Malte Helmert, University of Basel

Erez Karpas, Technion – Israel Institute of Technology

Carlos Linares López, Universidad Carlos III de Madrid

Gabriele Röger, University of Basel

Jordan Thayer, University of New Hampshire

Rong Zhou, PARC

Foreword

State-space search guided by heuristics, automatically derived from the problem representation, has been one of the most popular, and arguably one of the most successful, approaches to domain-independent planning in the last decade. While there has been significant developments in the design of planning heuristics, such that there are now many different kinds of heuristics, and some theories of how they relate to each other have begun to emerge, there is also a growing realisation that the search algorithm plays an equally important role in the approach. Recent work has highlighted some of the weaknesses of some search algorithms, but also the rich opportunities for exploiting synergies between the heuristic calculation and the search to improve both, drawing on the fact that domain-independent planning offers a declarative description of the state space (not just a "black box" successor function).

The workshop on Heuristics and Search for Domain-Independent Planning (HSDIP) follows in the tradition of the "Heuristics for Domain-Independent Planning (HDIP)" workshop series that was held at ICAPS 2007, 2009 and 2011. However, since the number contributions to the workshop relating to search (not just heuristics) has steadily grown, the ICAPS 2012 workshop widens the scope to explicitly encourage work on search for domain-independent planning.

Patrik Haslum, Malte Helmert, Erez Karpas, Carlos Linares López,
Gabriele Röger, Jordan Thayer, and Rong Zhou
HSDIP 2012 Organizers
June 2012

Contents

Deeply Preferred Operators: Lazy Search Meets Lookahead

Roei Bahumi and **Carmel Domshlak** and **Erez Karpas**
Faculty of Industrial Engineering & Management, Technion

Abstract

Heuristics in state-space search are primarily used to estimate the distance from states to the goal. In domain-independent heuristic-search planning, using extra information derived from the heuristic computation to mark some successors as preferred, and then biasing the search towards the preferred successors, resulted in significant improvements in planning performance. Preferred operators, however, help to discriminate only between the immediate successors of the evaluated state. We propose a simple and effective technique that takes advantage of more of the information provided by the heuristic computation. This technique, called *lazy lookahead*, consists of two components: A generalization of preferred operators to deeper descendants of the evaluated states, and a suitable generalization of deferred heuristic evaluation (aka lazy search) to such "deeply preferred" descendants. Our evaluation shows that employing lazy lookahead results in better performance than using standard preferred operators.

Introduction

Heuristic state-space search is one of the most prominent approaches to domain independent planning. For satisficing planning, the most common such approach is to use greedy best-first search, guided by heuristic functions. Heuristics are used primarily to estimate the distance from search states to the goal. The search algorithm can then use these distance estimates to choose a state which is likely to be closer to the goal, and thus hopefully to find a solution faster.

One of the most important advances in satisficing planning was the introduction of *helpful actions* in the FF planner (Hoffmann and Nebel 2001), where FF's relaxed plan was used not only for estimating the distance to the goal, but also to mark a few successors of the evaluated state as "helpful", in the sense of, "more likely to lead towards the goal". This concept was later generalized under the name of *preferred operators* (Helmert 2006), and was found especially helpful when used with *lazy search* (also called *deferred evaluation*, Richter and Helmert 2009). In lazy search, a state is evaluated not when it is generated, but when it is selected for expansion and removed from the open list. Lazy search aims at reducing the number of expensive heuristic evaluations: many states in the last layer of the search do not need to be evaluated, and preferred operators help focusing the evaluation on the more promising such states.

While which operators are considered preferred varies between the specific search components, the set of preferred operators of a given state always appears to be a subset of operators applicable at that state. At least in principle, this property appears to be unnecessarily limiting. Consider, for example, a state s for which a relaxed plan ρ^+ is generated. It is possible that ρ^+ (considered in terms of the original actions) is actually a valid plan from s to a goal state, and thus it provides us with the overall solution to the problem. Nevertheless, even using preferred operators, the search would need to evaluate at least every state along ρ^+ until the goal is found, despite the fact that we already have a solution at hand. Furthermore, it is possible that when the successor of s along ρ^+ is evaluated, a completely different relaxed plan will be generated for it, leading the search off the solution path that was already discovered.

Previous work has noted this, and suggested using FF's relaxed plan ρ^+ from state s for *lookahead* from s onwards (Vidal 2004). Specifically, an action sequence π applicable at state s is constructed from ρ^+, and then the state s' reached by π from s, called a lookahead state, is added to the open list, along with the regular successors of s. The weak point of such a lookahead is that often it is only some prefix of π that takes us closer to the goal, while the remaining part of π goes off in the wrong direction. In that case, adding only the end state achieved by π from s would probably not be the best thing to do.

Here we propose a technique that takes advantage of such *heuristically-suggested paths* π, that is, "preferred sequences" of real actions induced by the heuristic computation. This technique, called *lazy lookahead*, consists of two components. First, it extends the notion of preferred operators into what we call *deeply preferred operators*, which lead not only to the immediate successors of state s, but rather to all the states along the heuristically-suggested path π from s. Second, it extends the machinery of lazy search to properly support such deeply preferred descendants. Comparing to various approaches to satisficing heuristic-search planning that are based on standard notions of preferred operators and lazy search, our technique aims at reducing the number of heuristic evaluations even further, by obviating the need to compute heuristic estimates for states at various depths, not just in the last layer of the search. Our empirical evaluation shows that searching with lazy lookahead indeed results in

significant runtime and memory improvements, and even increases the number of planning tasks being solved.

Lazy Search Meets Lookahead

We consider planning tasks $\Pi = \langle P, A, cost, s_0, G\rangle$ formulated in STRIPS with action costs, where P are propositions, A are (standard syntax and semantics) actions, $s_0 \subseteq P$ is the inital state, and $G \subseteq P$ is the goal (Fikes and Nilsson 1971). The cost $cost(\pi)$ of an action sequence $\pi = \langle a_0, a_1, \ldots, a_n\rangle$ is $\sum_{i=0}^{n} cost(a_i)$. An action sequence π is an s-path if it is applicable in state s; the state resulting from applying an s-path π in s is denoted by $s[\![\pi]\!]$. An s-path is an s-plan if $G \subseteq s_0[\![\pi]\!]$. In basic satisficing planning, the objective is to find an s_0-plan as efficiently as possible.

Lazy Lookahead

As previously mentioned, heuristics can provide us with more than just an estimate of the distance from state s to the goal. For now, let us assume that an (imperfect) oracle provided us with some s-path $\pi = \langle a_1, a_2, \ldots a_n\rangle$, which is likely to lead towards the goal. We call such a path π a *heuristically-suggested path*, and later we discuss usage of a distinct class of heuristics as a respective oracle. In any case, given such π, the current mechanisms for supporting preferred operators allows us to "favor" the immediate successor $s[\![\langle a_1\rangle]\!]$ of s, biasing the search towards expanding it. However, these mechanisms cannot assist us with "favoring" the indirect successors $\{s[\![\langle a_1, \ldots, a_i\rangle]\!]\}_{i=2}^{n}$ of s, despite their purported attractiveness suggested via π.

Aiming at taking advantage of as much of the information provided by the heuristic computation as possible, we propose a simple new mechanism for preferring "deeper" states along the simulated execution of the heuristically-suggested paths that we refer to as *lazy lookahead*. Given a heuristically-suggested s-path $\pi = \langle a_1, \ldots, a_n\rangle$, lazy lookahead adds to the open list *all* the states $\{s[\![\langle a_1, \ldots, a_i\rangle]\!]\}_{i=1}^{n}$, with the lazy heuristic estimate of $s[\![\langle a_1, \ldots, a_i\rangle]\!]$ being set to the true heuristic value of s, adjusted by the cost of the actions $\{a_1, \ldots, a_i\}$. That is, for $1 \leq i \leq n$, $h^{\text{lazy}}(s[\![\langle a_1, \ldots, a_i\rangle]\!]) = h(s) - \sum_{j=1}^{i} cost(a_j)$.

Note that the adjusted heuristic estimate differs from what is normally done in lazy search, where the successors of state s are added to the open list with a heuristic estimate of $h(s)$ (Richter and Helmert 2009). The reason for this difference is that we want the "deeper" states to be expanded earlier, as they are more likely to be closer to a goal state. In contrast, lazy search only adds states on the same level, making this argument irrelevant. We also note that it is, of course, possible to simply perform an eager lookahead by evaluating each state along the simulated execution of π at s before adding these states to the open list. However, as heuristic computation is typically much more expensive than state generation, we believe this effort is very unlikely to pay off. For example, if the last state $s[\![\pi]\!]$ along that simulated execution is already a goal state, then search with lazy lookahead finishes immediately, with no need to evaluate all the intermediate states $\{s[\![\langle a_1, \ldots, a_i\rangle]\!]\}_{i=1}^{n-1}$. Even if the last state $s[\![\pi]\!]$ is not a goal state, but it has a lower true heuristic estimate than $h(s)$, the search will continue from there, again, without the need to evaluate all the intermediate states.

Of course, there is no guarantee that a heuristically-suggested path leads anywhere near the goal. Consider the following scenario, where the search reaches a large heuristic plateau: State s is evaluated, and a heuristically-suggested path leading to state s' is generated. s' is evaluated next and found to have the same heuristic estimate as state s, and a heuristically-suggested path leading to state s'' is generated, and so on. This adds numerous states to the open list, as each heuristically-suggested path adds all the states along its simulation to the open list. In this scenario, the open list fills up with "junk" states, which might go much deeper than what search without lookahead would need to expand to escape the bad region. Having this potential negative effect of the lookahead in mind, we have also evaluated a variant of the lazy lookahead that we call *conditional lookahead*. The basic idea is simply to apply lookahead only for a selection of the expanded states. A simple and intuitive selection condition we have evaluated empirically aimed at preventing sequential lookahead that does not show any improvement. Specifically, we only look ahead from state s if it meets one of the following criteria:

1. s has been generated via the "regular" search, not via a heuristically-suggested path.
2. s was generated via a heuristically-suggested path from state s', and $h(s) < h(s')$.

These conditions prevent us from performing deeper and deeper lookahead, without any sign of improvement. Of course, more involved conditions, which might also be based upon information from the heuristic h, can be found even more beneficial in practice.

Generating Heuristically-Suggested Paths

Having described the way we suggest exploiting heuristically-suggested paths in the context of best-first search, we proceed with considering means for generating such heuristically-suggested paths. A natural option that we adopt here is to extract heuristically-suggested paths from the artifacts of computation of a certain class of heuristics—those that are based on solving a simplified version of the planning task at hand. Examples of such heuristics include FF's relaxed plan heuristic (Hoffmann and Nebel 2001), and abstraction heuristics such as PDBs (Culberson and Schaeffer 1998), merge and shrink (Helmert, Haslum, and Hoffmann 2008), and implicit abstraction heuristics (Katz and Domshlak 2010). What all of these heuristics have in common is that the heuristic value for state s is based upon a solution to a simpler, but still a planning, problem. While the state-of-the-art abstraction heuristics listed above avoid storing these solutions explicitly in order to reduce their memory overhead, the relaxed-plan FF heuristic generates such a solution every time it is evaluated. Note that this solution does not have to be a linear sequence of actions, but can rather comprise a partially ordered set of actions. The procedure we describe next is general, and can be used

with any heuristic that is based upon estimating the cost of such a partially ordered set of actions.

Given a partially ordered relaxed plan ρ^+ from state s, we attempt to find an applicable action sequence π which is compatible with that partially ordered plan. Our procedure attempts to apply actions from ρ^+, while keeping track of the current heuristically-suggested path, and of the state that is reached by it. An action from ρ^+ is eligible to be tried if all of its predecessors in ρ^+ have already been added to π. The partially ordered plan ρ^+ is traversed according to some linear order compatible with it, and checks whether an eligible action is applicable at the state $s[\![\pi]\!]$ reached by the current heuristically-suggested path. If an applicable action a was found, we update the currently reached state to $s[\![\pi \cdot \langle a \rangle]\!]$, and we update π to $\pi \cdot \langle a \rangle$. Once a complete pass over ρ^+ is accomplished, we go back to the beginning of ρ^+, and perform another pass, as some previously inapplicable actions might now become applicable. The procedure terminates after a complete pass over ρ^+, in which no action was added to the heuristically-suggested path π. Note that traversing ρ^+ according to different orders might lead to different heuristically-suggested path.

It is possible to employ some sophisticated tactics for choosing the linear order in which the partially ordered plan ρ^+ shall be traversed; for instance, some of such possible tactics in the context of FF relaxed plans are discussed in a related context by Vidal (2004). In our evaluation, however, our objective was to separate between the basics and optimizations, and thus we have implemented two simple choices: either try actions according to some arbitrary, implementation-dependent order, or choose the next eligible action at random. The only optimization we do apply is using the layer structure of FF's relaxed plan so that the linear order on actions is compatible with the order of the layers, and choosing at random from actions in the same layer. As the empirical evaluation will demonstrate, even these simple choices suffice to improve over the baseline lazy search with the FF heuristic.

Empirical Evaluation

In order to evaluate lazy lookahead empirically, we implemented it on top of the Fast Downward planning system (Helmert 2006), and conducted experiments on all domains from IPC 1998–2008, except MOVIE and ASSEMBLY. All of the experiments reported here were conducted on a single core of an Intel E8400 CPU, with a time limit of 30 minutes, and a memory limit of 1.5 GB.

As the current implementations of abstraction heuristics do not support obtaining a concrete solution for the abstraction from each state, we only evaluated the effect of lazy lookahead on search using FF's relaxed plan heuristic. In all of the experiments here, we used lazy greedy best-first search (with lazy lookahead, in our configurations) using boosted dual queues (Richter and Helmert 2009) and preferred operators (deeply preferred operators in our configurations). All of the states which were generated by the lazy lookahead procedure have been distinguished as preferred. We compare the baseline relaxed plan heuristic with using both (unconditional) lazy lookahead (denoted by *LL*) and conditional lazy lookahead (denoted by *CLL*). For each such method, we have two variants of partial order plans linearization: random (denoted with the prefix *rnd-*) and arbitrary (no prefix).

domain	rnd-LL	LL	rnd-CLL	CLL	FF
airport (50)	**38**	37	35	**38**	37
depot (22)	**20**	19	18	19	19
elevators (30)	**30**	19	25	12	11
logistics98 (35)	**35**	33	**35**	**35**	33
mystery (30)	**16**	15	**16**	**16**	**16**
openstacks (30)	6	6	6	6	6
optical-telegraphs (48)	**3**	**3**	**3**	**3**	2
parcprinter (30)	17	**27**	18	26	21
pathways (30)	22	20	22	21	**29**
philosophers (48)	**48**	**48**	40	20	42
pw-notankage (50)	**44**	43	43	43	41
pw-tankage (50)	**43**	41	41	40	40
psr-large (50)	**16**	15	**16**	15	15
psr-middle (50)	42	43	**44**	43	42
schedule (150)	**150**	**150**	**150**	**150**	149
sokoban (30)	**29**	28	28	28	28
storage (30)	19	17	19	17	**20**
transport (30)	**30**	**30**	**30**	**30**	21
trucks (30)	17	16	17	16	**18**
woodworking (30)	29	**30**	**30**	**30**	27
TOTAL (shown domains)	**654**	640	636	608	617

Table 1: Number of tasks solved, per domain and overall. Domains where all approaches solved the same number of problems are not shown.

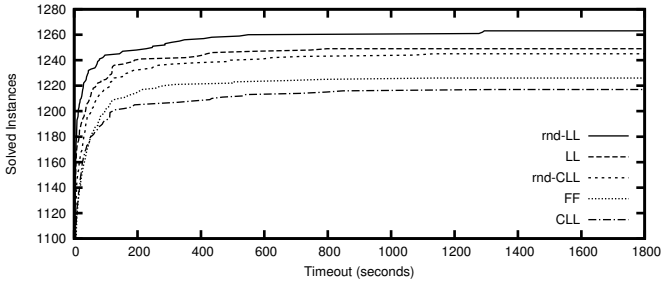


Figure 1: Anytime profile of different approaches. Each line shows the number of problems solved by each approach (y-axis), under different timeouts (x-axis).

Table 1 shows the number of problems solved using each approach, in each domain. We omitted here the domains in which all the approaches solved all the tasks in the domain. These results show that greedy best first search using the randomized variant of both conditional and unconditional lazy lookahead solves overall more problems than the baseline, and that adopting lazy lookahead was beneficial on more domains than domains in which it hurt the performance.

A more detailed examination of the results shows that randomization in the lookahead greatly helps in ELEVATORS, and greatly hurts in PARCPRINTER. One possible reason for this is that the arbitrary action ordering in PARCPRINTER is actually very good for the relaxed plan, while in ELEVATORS it is very bad. However, overall, randomization performs better than using the arbitrary order.

Another observation is that conditional lookahead does not seem to pay off, and does not seem to perform as well as (unconditional) lookahead. This is likely to do with the extra overhead associated with conditional lookahead, where we

domain	rnd-LL	LL	rnd-CLL	CLL	FF
airport	0.81	0.63	**0.82**	0.65	0.72
blocks	**0.71**	0.7	0.69	0.58	0.7
depot	0.59	0.55	**0.63**	0.55	0.36
driverlog	0.7	0.68	**0.72**	0.71	0.32
elevators	**0.93**	0.35	0.46	0.12	0.11
freecell	**0.76**	0.74	0.65	0.66	0.37
grid	**0.87**	0.59	0.62	0.62	0.44
gripper	0.76	0.88	0.93	**0.94**	0.22
logistics00	**0.82**	0.7	0.78	0.78	0.1
logistics98	**0.83**	0.51	0.8	0.58	0.06
miconic	0.83	**0.95**	0.77	0.91	0.17
mprime	**0.81**	0.72	0.58	0.62	0.29
mystery	0.66	0.71	**0.81**	0.73	0.48
openstacks	0.53	0.52	0.57	0.44	**0.76**
optical-telegraphs	0.51	**0.72**	0.17	0.04	0.07
parcprinter	0.46	**0.65**	0.5	0.59	0.49
pathways	**0.89**	0.82	0.8	0.8	0.12
pegsol	0.47	0.55	0.6	0.56	**0.71**
philosophers	0.82	**1**	0.02	0.03	0.04
pw-notankage	0.48	0.53	**0.58**	0.57	0.55
pw-tankage	**0.59**	0.56	0.48	0.45	0.39
psr-large	0.82	0.84	0.88	0.87	**0.91**
psr-middle	0.84	0.84	0.88	0.89	**0.91**
psr-small	0.66	0.67	0.73	0.73	**1**
rovers	0.84	0.81	**0.87**	0.85	0.07
satellite	0.76	0.87	0.82	**0.9**	0.12
scanalyzer	0.78	**0.85**	0.5	0.48	0.29
schedule	0.73	0.69	**0.74**	0.67	0.11
sokoban	0.72	0.62	0.72	0.72	**0.85**
storage	0.64	0.62	**0.82**	0.78	0.81
tpp	**0.93**	0.59	0.48	0.32	0.14
transport	**0.83**	0.75	0.36	0.27	0.16
trucks	0.46	0.38	**0.69**	0.46	0.52
woodworking	0.74	0.8	0.75	**0.83**	0.42
zenotravel	0.83	**0.87**	0.75	0.86	0.2
NORM. AVG	**0.73**	0.69	0.66	0.62	0.4

Table 2: Generated states: average metric scores.

domain	rnd-LL	LL	rnd-CLL	CLL	FF
airport	0.84	0.66	**0.85**	0.7	0.53
blocks	**0.75**	**0.75**	0.69	0.59	0.57
depot	0.61	0.56	**0.62**	0.54	0.32
driverlog	0.7	0.68	**0.72**	0.7	0.28
elevators	**0.93**	0.35	0.44	0.12	0.1
freecell	**0.75**	**0.75**	0.62	0.66	0.3
grid	**0.87**	0.59	0.56	0.58	0.3
gripper	0.76	0.88	**0.94**	0.92	0.11
logistics00	**0.79**	0.68	0.75	0.77	0.07
logistics98	**0.83**	0.51	0.8	0.59	0.06
miconic	0.83	**0.95**	0.77	0.91	0.15
mprime	**0.82**	0.72	0.61	0.65	0.32
mystery	0.7	0.73	**0.8**	0.73	0.47
openstacks	0.62	0.6	0.59	0.48	**0.68**
optical-telegraphs	0.51	**0.74**	0.14	0.03	0.04
parcprinter	0.41	**0.59**	0.48	0.56	0.38
pathways	**0.92**	0.83	0.76	0.79	0.15
pegsol	0.53	**0.63**	**0.63**	0.57	0.59
philosophers	**1**	**1**	0.02	0.02	0.02
pw-notankage	0.52	0.56	**0.59**	0.58	0.5
pw-tankage	**0.59**	0.58	0.47	0.46	0.37
psr-large	0.89	0.9	**0.93**	0.92	0.9
psr-middle	0.9	0.91	**0.93**	**0.93**	0.91
psr-small	0.98	0.98	**1**	0.99	0.97
rovers	0.84	0.85	0.85	**0.88**	0.06
satellite	0.77	0.87	0.82	**0.9**	0.11
scanalyzer	0.79	**0.85**	0.5	0.47	0.26
schedule	**0.74**	0.68	0.73	0.66	0.12
sokoban	**0.87**	0.79	0.79	0.8	0.7
storage	0.76	0.74	**0.86**	0.82	0.77
tpp	**0.91**	0.57	0.42	0.3	0.07
transport	**0.83**	0.75	0.34	0.25	0.13
trucks	0.42	0.35	**0.66**	0.46	0.58
woodworking	0.74	0.8	0.71	**0.82**	0.45
zenotravel	0.79	0.82	0.74	**0.85**	0.19
NORM. AVG	**0.76**	0.72	0.66	0.63	0.36

Table 3: Evaluated states: average metric scores.

need to keep track of how each state was reached (via lookahead or not), and of the heuristic value of the state where the respective lookahead started (if it was reached via lookahead).

However, the number of problems solved after 30 minutes does not tell the complete tale. Figure 1 shows the number of problems solved by each approach, under different timeouts. As the results show, deeply preferred operators help, no matter which timeout is used, with an even greater advantage over the baseline relaxed plan heuristic when the timeout is smaller.

Finally, we wish to explore the impact of using deeply preferred operators on the number of generated states, as well as on the number of heuristic evaluations performed. Tables 2 and 3 give the metric score of the number of generated and evaluated state, respectively, over the problems solved by all 5 configurations. The metric score for configuration c on some problem is v^*/v_c, where v_c is the value of configuration c (number of generated or evaluates stated), and v^* is the best value of any configuration on that problem. Thus the best value for each problem is assigned a metric score of 1, and generating twice as many states would lead to a score of 0.5. For each domain we report the average score, as well as the average across all domain averages. As these tables show, using lazy lookahead significantly reduces both the number of generated states, as well as the number of evaluated states, and this across all the four settings of lazy lookahead.

References

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

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *AIJ* 2:189–208.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2008. Explicit-state abstraction: A new method for generating heuristic functions. In *Proc. AAAI 2008*, 1547–1550.

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

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

Katz, M., and Domshlak, C. 2010. Implicit abstraction heuristics. *JAIR* 39:51–126.

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

Vidal, V. 2004. A lookahead strategy for heuristic search planning. In *Proc. ICAPS 2004*, 150–159.

Width and Serialization of Classical Planning Problems*

Nir Lipovetzky
Universitat Pompeu Fabra
Barcelona, Spain
nir.lipovetzky@upf.edu

Hector Geffner
ICREA & Universitat Pompeu Fabra
Barcelona, SPAIN
hector.geffner@upf.edu

Abstract

We introduce a width parameter that bounds the complexity of classical planning problems and domains, along with a simple but effective blind-search procedure that runs in time that is exponential in the problem width. We show that many benchmark domains have a bounded and small width provided that *goals* are restricted to *single atoms*, and hence that such problems are provably solvable in low polynomial time. We then focus on the practical value of these ideas over the existing benchmarks which feature conjunctive goals. We show that the blind-search procedure can be used for both serializing the goal into subgoals and for solving the resulting problems, resulting in a 'blind' planner that competes well with a best-first search planner guided by state-of-the-art heuristics. In addition, ideas like helpful actions and landmarks can be integrated as well, producing a planner with state-of-the-art performance.

Introduction

Various approaches have been developed for explaining the gap between the complexity of planning (Bylander 1994), and ability of current planners to solve most existing benchmarks in a few seconds (Hoffmann and Nebel 2001; Richter and Westphal 2010). *Tractable planning* has been devoted to the identification of planning fragments that due to syntactic or structural restrictions can be solved in polynomial time; fragments that include for example problems with single atom preconditions and goals, among others (Bylander 1994; Bäckström 1996). *Factored planning* has appealed instead to mappings of planning problems into Constraint Satisfaction Problems, and the notion of *width* over CSPs (Amir and Engelhardt 2003; Brafman and Domshlak 2006). The width of a CSP measures the number of variables that have to be collapsed to ensure that the graph underlying the CSP becomes a tree (Freuder 1982; Dechter 2003). The complexity of a CSP is exponential in the problem width. A notion of width for classical planning using a form of Hamming distance was introduced in (Chen and Giménez 2007), where the distance is set to the number of problem variables whose value needs to be changed in order to increase the number of achieved goals. These proposals, however, do not appear to explain the apparent simplicity of the standard domains.

A related thread of research has aimed at understanding the performance of modern heuristic search planners by analyzing the characteristics of the optimal delete-relaxation heuristic h^+ that planners approximate for guiding the search for plans (Hoffmann 2005; 2011). For instance, the lack of local minima for h^+ implies that the search for plans (and hence the global minimum of h^+) can be achieved by local search, and this local search is tractable when the distance to the states that decrement h^+ is bounded by a constant. This type of analysis has shed light on the characteristics of existing domains where heuristic search planning is easy, although it doesn't address explicitly the conditions under which the heuristic h^+ is easy to compute, nor whether it's the use of this heuristic that makes these domains easy.

The aim of this paper is to explore a new width notion for planning that can be useful both theoretically and practically. More precisely, the contributions of the paper are:

1. a new width notion for planning problems and domains;
2. a proof that many of the existing domains have a bounded and low width when goals are restricted to single atoms;
3. a simple, blind-search planning algorithm (*IW*) that runs in time exponential in the problem width;
4. a blind-search planner that uses *IW* for serializing a problem into subproblems and for solving the subproblems, which is competitive with a best-first search planner using state-of-the-art heuristics;
5. a state-of-the-art planner that integrates new ideas and old.

The organization of the paper follows this structure, preceded by a review of basic notions in planning.

Planning

The classical model for planning $\mathcal{S} = \langle S, s_0, S_G, A, f\rangle$ is made up of a finite set of states S, an initial state s_0, a set of goal states S_G, and actions $a \in A(s)$ that deterministically map one state s into another $s' = f(a, s)$, where $A(s)$ is the set of actions applicable in the state s. The solution to a classical planning model is a sequence of actions $a_0, \ldots, a_m$

*To appear at ECAI-2012.

that generates a state sequence $s_0, s_1, \ldots, s_{m+1}$ such that $a_i \in A(s_i)$, $s_{i+1} = f(a_i, s_i)$, and $s_{m+1} \in S_G$.

A classical planning problem P defines a classical model in compact form through a set of variables. We assume a Strips language $P = \langle F, I, O, G \rangle$, where F is the set of boolean variables, atoms, or fluents, I is the set of atoms characterizing the initial state, O is the set of actions, and G is the set of goal atoms. All definitions below extend easily to other planning languages provided that states are valuations over a set of variables.

We assume that *action costs* are all 1, so that plan cost is plan length, and the *optimal* plans are the *shortest* ones. We write $P(t)$ to denote the planning problem that is like P but with goal t. The cost of the goal t in P is the cost of an optimal plan for $P(t)$.

Width

A state s is reachable from the initial state $s_0 = I$ in P, if there is a state trajectory $s_0, s_1, \ldots, s_n$, such that s_{i+1} is a successor of s_i for some action a_i, and $s = s_n$. Since this reachability relation is exponential in the number of atoms, we define a different reachability relation over *tuples (conjunctions) of atoms* t of bounded size. The key question is when a tuple t' can be regarded as a 'successor' of a tuple t in P. If this is taken to represent the presence of an action a in P such that the *regression* of t through a is in t', the reachability relation on tuples ends up being too weak. A stronger reachability relation on tuples can be obtained if we assume that both t and t' are achieved *optimally*. We will define indeed that t' is a successor of t if *every optimal plan* for t can be extended into an *optimal plan* for t' by just adding one action. In this way, the 'side-effects' of the optimal plans for t can be used to achieve t' optimally from t. This is formalized below in terms of *tuple graphs*, where T^i stands for the collection of tuples from P with size no greater than a given positive integer i:

Definition 1 *For $P = \langle F, I, O, G \rangle$, $\mathcal{G}^i$ is the* graph *with vertices from T^i defined inductively as follows:*

1. *t is a root vertex in $\mathcal{G}^i$ iff t is true in I,*
2. *$t \rightarrow t'$ is a directed edge in $\mathcal{G}^i$ iff t is in $\mathcal{G}^i$ and for every optimal plan π for $P(t)$ there is an action $a \in O$ such that π followed by a is an optimal plan for $P(t')$.*

In other words, the presence of the tuple t' of at most i atoms in the graph $\mathcal{G}^i$ indicates that either t' is true in I or that there is another tuple t of at most i atoms in $\mathcal{G}^i$ such that *all* the optimal plans π for t yield optimal plans for t', once a suitable action a is appended to π.

The graph $\mathcal{G}^i$ is acyclic because an edge $t \rightarrow t'$ implies that the optimal cost for achieving t' is the optimal cost of achieving t plus 1. Since we are associating plan cost with plan length, this means also that a tuple at depth k in the graph has optimal cost k.

Let us now say that a goal formula G_1 *implies* goal formula G_2 *in a problem* P, if all the *optimal plans* for G_1 are also *optimal plans* for G_2. Notice that this is not the standard logical implication that requires the formula $G_1 \supset G_2$ to be true in all reachable states, and hence, an invariant. For example, the goal $G_1 = \{hold(b)\}$ implies the goal $G_2 = \{clear(a)\}$ in *Blocks World* if $on(b, a)$ is true initially, yet $hold(b) \supset clear(a)$ is not an invariant.

Provided with this notion of implication, we define the *width* of a planning problem P and, more generally, the *width* of an arbitrary goal formula ϕ relative to P as follows:

Definition 2 *For a formula ϕ over the fluents in P that is not true in the initial situation I, the* width *of ϕ relative to P is the min w such that $\mathcal{G}^w$ contains a tuple that implies ϕ. If ϕ is true in I, its width is 0.*

Definition 3 *The* width *of a planning problem P, $w(P)$, is the* width *of its goal G relative to P.*

As in the case of graphical models, the width of a problem give us a bound on the complexity of solving the problem:

Theorem 4 *If $w(P) = i$, P can be solved optimally in time that is exponential in i.*

Of course, the crucial question is whether there are interesting planning problems that have a bounded, and hopefully small width. The answer is yes. Indeed, *most domain benchmarks appear to have a small width independent of the size of the problems, provided that the goal G is restricted to a single atom*. Of course, this result doesn't settle the complexity of the existing benchmarks where goals are *not* single atoms, yet as far as we know, it's the first formal result that places the complexity of these benchmarks squarely on the *goal structure*, and not on the *domain structure*. That is, if a domain has a low width when goals are single atoms, then when a domain instance is not easy, it can only be due to conjunctive goals.

We state the result about the width of a few domains. For most other benchmark domains, the same result seems to hold, but we have not carried out the proofs. Later on, however, we will report experiments that bear on this issue.

Theorem 5 *The domains Blocks, Logistics, and n-puzzle have a* bounded width *independent of the problem size and initial situation, provided that the goals are restricted to* single atoms.

It turns out indeed that for single atom goals, the width of Blocks, Logistics and n-puzzle is at most 2. We omit the proofs for lack of space, but for illustration purposes we prove $w(G) = 1$ for any goal $G = ontable(b)$ in Blocks.

Clearly, if G is true in the initial situation, the tuple G will belong to the graph $\mathcal{G}^i$ for $i = 1$. Thus, assume that G is not true initially, and let $b_1, \ldots, b_{n-1}$ be the blocks on top of b, starting from the top, and let $b = b_n$. We can then just show that the path

$$clear(b_1), hold(b_1), ontable(b_1), hold(b_2), \ldots, ontable(b_n)$$

makes it into $\mathcal{G}^1$. For a path $t_0, t_1, \ldots, t_n$ to be in $\mathcal{G}^1$ when t_0 is true in the initial situation, we just need to show that any optimal plan for t_i can be extended by means of a single action into an optimal plan for t_{i+1}, $i = 0, \ldots, n-1$. This is trivial in this case, as the optimal plans for $hold(b_i)$ can always be extended with the action $putdown(b_i)$ into optimal plans for $ontable(b_i)$, while the *optimal plans* for

$ontable(b_i)$ from the above initial situation can all be extended with the action $unstack(b_{i+1}, b_{i+2})$ into *optimal plans* for $hold(b_{i+1})$. It's important to notice that without the restriction to *optimal plans* this reasoning would not get through.

Basic Algorithm: Iterated Width Search

We turn now to the planning algorithm that achieves the complexity bounds expressed by Theorem 4. The algorithm, called *Iterated Width* search or *IW*, consists of a sequence of calls *IW*(i) for $i = 0, 1, 2, \ldots$ over a problem P until the problem is solved. Each iteration *IW*(i) is an i-width search that is complete for problems whose width is bounded by i and has complexity is $O(n^i)$, where n is the number of problem variables. If P is solvable and its width is w, *IW* will solve P in at most $w + 1$ iterations with a complexity $O(n^w)$. *IW*(i) is *a plain forward-state breadth-first search* with just one change: right after a state s is generated, the state is pruned if it doesn't pass a simple *novelty test* that depends on i.

Definition 6 *A newly generated state s* produces a new tuple of atoms t *iff s is the first state generated in the search that makes t true. The size of the smallest new tuple of atoms produced by s is called the* novelty *of s. When s does not generate a new tuple, it's novelty is set to $n + 1$ where n is the number of problem variables.*

In other words, if s is the first state generated in all the search that makes an atom p true, its novelty is 1. If s does not generate a new atom but generates a new pair (p, q), its novelty is 2, and so on. Likewise, if s does not generate a new tuple at all because the same state has been generated before, then its novelty is set to $n + 1$. The higher the novelty measure, the less novel the state. The iterations *IW*(i) are plain *breadth-first searches* that treat newly generated states with novelty measure greater than i as if they were 'duplicate' states:

Definition 7 *IW*(i) *is a breadth-first search that prunes newly generated states when their novelty measure is greater than i.*

Notice that *IW*(n), when n is the number of atoms in the problem, just prunes truly duplicate states and it is therefore complete. On the other hand, *IW*(i) for lower i values prunes many states and is not. Indeed, the number of states *not* pruned in *IW*(1) is $O(n)$ and similarly, the number of states not pruned in *IW*(i) is $O(n^i)$. Likewise, since the novelty of a state is never 0, *IW*(0) prunes all the children states of the initial state s_0, and thus *IW*(0) solves P iff the goal is true in the initial situation. The resulting planning algorithm *IW* is just a series of i-width searches *IW*(i), for increasing values of i:

Definition 8 *Iterated Width (IW) calls IW(i) sequentially for $i = 0, 1, 2, \ldots$ until the problem is solved or i exceeds the number of problem variables.*

Iterated Width (*IW*) is thus a *blind-search algorithm* similar to Iterative Deepening (*ID*) except for two differences. First, each iteration is a pruned *depth-first* search in *ID*, and a pruned *breadth-first* search in *IW*. Second, each iteration increases pruning depth in *ID*, and pruning width or novelty in *IW*.

From the considerations above it is straightforward to show that *IW* like *ID* is *sound* and *complete.* On the other hand, while *IW*(w) is *optimal* for a problem P of width w, *IW* is not necessarily so. The reason is that *IW* may solve P in an iteration *IW*(i) for i smaller than w.[1]

Domain	I	$w_e = 1$	$w_e = 2$	$w_e > 2$
8puzzle	400	55%	45%	0%
Barman	232	9%	0%	91%
Blocks World	598	26%	74%	0%
Cybersecure	86	65%	0%	35%
Depots	189	11%	66%	23%
Driver	259	45%	55%	0%
Elevators	510	0%	100%	0%
Ferry	650	36%	64%	0%
Floortile	538	96%	4%	0%
Freecell	76	8%	92%	0%
Grid	19	5%	84%	11%
Gripper	1275	0%	100%	0%
Logistics	249	18%	82%	0%
Miconic	650	0%	100%	0%
Mprime	43	5%	95%	0%
Mystery	30	7%	93%	0%
NoMystery	210	0%	100%	0%
OpenStacks	630	0%	0%	100%
OpenStacksIPC6	1230	5%	16%	79%
ParcPrinter	975	85%	15%	0%
Parking	540	77%	23%	0%
Pegsol	964	92%	8%	0%
Pipes-NonTan	259	44%	56%	0%
Pipes-Tan	369	59%	37%	3%
PSRsmall	316	92%	0%	8%
Rovers	488	47%	53%	0%
Satellite	308	11%	89%	0%
Scanalyzer	624	100%	0%	0%
Sokoban	153	37%	36%	27%
Storage	240	100%	0%	0%
Tidybot	84	12%	39%	49%
Tpp	315	0%	92%	8%
Transport	330	0%	100%	0%
Trucks	345	0%	100%	0%
Visitall	21859	100%	0%	0%
Woodworking	1659	100%	0%	0%
Zeno	219	21%	79%	0%
Summary	37921	37.0%	51.3%	11.7%

Table 1: Effective width of single goal instances obtained from existing benchmarks by splitting problems with N atomic goals into N problems with single goals. I is number of resulting instances. The other columns show percentage of instances with effective width 1, 2, or greater.

[1] As an illustration, the goal G of width 2 is achieved non-optimally by *IW*(1) when $I = \{p_1, q_1\}$ and the actions are $a_i : p_i \rightarrow p_{i+1}$ and $b_i : q_i \rightarrow q_{i+1}$ for $i = 1, \ldots, 5$, along with $b : p_6 \rightarrow G$ and $c : p_3, q_3 \rightarrow G$. Indeed, *IW*$(2)$ achieves G optimally at cost 5 using the action c, yet this action is never applied in *IW*(1), where states that result from applying the actions a_i when q_j is true for $j > 1$ are pruned, and states that result from

Nonetheless the completeness and optimality of *IW*(w) for problems with width w provides the right complexity bound for *IW*:

Theorem 9 *For solvable problems P, the time and space complexity of IW are exponential in $w(P)$.*

It's important to realize that this bound is achieved without knowing the actual width of P. This follows from the result below, whose proof we omit for lack of space:

Theorem 10 *For a solvable problem P with width w, IW(w) solves P optimally in time exponential in w.*

The algorithm *IW*(w) is guaranteed to solve P if $w(P) = w$, yet as discussed above, the algorithm *IW* does not assume that this width is known and thus makes the *IW*(i) calls in order starting from $i = 0$. We refer to the min value of i for which *IW*(i) solves P as the *effective width* of P, $w_e(P)$, which is never higher than the real width $w(P)$.

The effective width $w_e(P)$ provides an approximation of the actual width $w(P)$. While proving formally that most benchmark domains have bounded width *for single atom goals* is tedious, we have run the algorithm *IW* to compute the *effective width* of such goals. The results are shown in Table 1 . We tested domains from previous IPCs. For each instance with N goal atoms, we created N instances with a single goal, and run *IW* over each one of them. The total number of instances is 37921. For each domain we show the total number of single goal instances, and the percentage of instances that have effective widths w_e equal to 1, 2, or greater than 2. The last row in the table shows the average percentage over all domains: 37% with $w_e = 1$, 51% with $w_e = 2$, and less than 12% with $w_e > 2$. *That is, on average, less than 12% of the instances have effective width greater than 2.* Actually, in most domains *all* the instances have effective width at most 2, and in four domains, all the instances have effective width 1. The instances with a majority of atomic goals with an effective width greater than 2 are from the domains Barman, Openstacks, and Tidybot (the first and last from the 2011 IPC).

Iterated Width (*IW*) is a complete blind-search algorithm like Iterative Deepening (*ID*) and Breadth-First Search (*BrFS*). We have also tested the three algorithms over the set of 37921 single goal instances above. The results are shown in Table 2 . With memory and time limits of 2GB and 2h, *ID* and *BrFS* solve less than 25% of the instances, while *IW* solves more than 90%, which is almost as many as a Greedy Best First Search guided by the additive heuristic (also shown in the Table). The result suggests that *IW* manages to exploit the low width of these problems much better than the other blind-search algorithms. We will see below that a simple extension suffices to make *IW* competitive with a *heuristic* planner over the standard benchmark instances that feature *joint goals*.

applying the actions b_i when p_j is true for $j > 1$ are pruned too. As a result, *IW*(1) prunes the states with pairs such as (p_3, q_2) and (p_2, q_3), and does not generate states with the pair (p_3, q_3), which are required for reaching G optimally. *IW*(1) however reaches G at the non-optimal cost 7 using the action b.

# Instances	*IW*	*ID*	*BrFS*	*GBFS* + h_{add}
37921	34627	9010	8762	34849

Table 2: Blind-search algorithm *IW* compared with two other blind-search algorithms: Iterative Deepening (*ID*) and Breadth-First Search (*BrFS*). Numbers report coverage over benchmark domains with single atomic goals. Also included for comparison the figure for heuristic Greedy Best First Search (*GBFS*) with h_{add}.

Serialization

The fact that single goal atoms can be achieved quite effectively in most benchmarks domains by a pruned breadth-first search that does not look at the goal in any way, suggests that the complexity of benchmarks comes from conjunctive goals. Indeed, this has been the intuition in the field of planning since its beginnings where goal decomposition was deemed as a crucial and characteristic technique. The analysis above formalizes this intuition by showing that the effective width of single atom goals in existing benchmarks is low. This old intuition also suggests that the power of planners that can handle single goals efficiently can be exploited for conjunctive goals through some form of decomposition.

Serialized Iterated Width is a search algorithm that uses the iterated width searches both for constructing a serialization of the problem $P = \langle F, I, O, G\rangle$ and for solving the resulting subproblems. While *IW* is a sequence of i-width searches *IW*(i), $i = 0, 1, \ldots$ over the same problem P, *SIW* is a sequence of *IW* calls over $|G|$ subproblems P_k, $k = 1, \ldots, |G|$. The definition of *SIW* takes advantage of the fact that *IW* is a blind-search procedure that doesn't need to know the goals of the problem in advance; it just needs to recognize them in order to stop. Thanks to this feature *IW* is used both for decomposing P into the sequence of subproblems P_k and for solving each one of them. The plan for P is the concatenation of the plans obtained for the subproblems.

Definition 11 *Serialized Iterated Width (SIW) over $P = \langle F, I, O, G\rangle$ consists of a sequence of calls to IW over the problems $P_k = \langle F, I_k, O, G_k\rangle$, $k = 1, \ldots, |G|$, where*

1. $I_1 = I$,
2. G_k *is the first* consistent *set of atoms achieved from I_k such that $G_{k-1} \subset G_k \subseteq G$ and $|G_k| = k$; $G_0 = \emptyset$*
3. I_{k+1} *represents the state where G_k is achieved, $1 < k < |G|$.*

In other words, the k-th subcall of *SIW* stops when *IW* generates a state s_k that consistently achieves k goals from G: those achieved in the previous subcall and a new goal from G. The same is required from the next subcall that starts at s_k. The state s_k *consistently* achieves $G_k \subseteq G$ if s_k achieves G_k, and G_k does not need to be undone in order to achieve G. This last condition is checked by testing whether $h_{max}(s_k) = \infty$ is true in P once the actions that delete atoms from G_k are excluded (Bonet and Geffner 2001). Notice that *SIW* does not use heuristic estimators to the goal, and does not even know what goal G_k is when *IW* is invoked on subproblem P_k: it finds this out when *IW* generates a set of atoms G' such that $G_{k-1} \subset G' \subseteq G$ and

		Serialized *IW* (*SIW*)				GBFS + h_{add}		
Domain	I	S	Q	T	M/Aw_e	S	Q	T
8puzzle	50	50	**42.34**	0.64	4/1.75	50	55.94	**0.07**
Barman	20	1	–		–	–	–	–
Blocks World	50	50	**48.32**	5.05	3/1.22	50	122.96	**3.50**
Cybersecure	30	–	–		–	–	–	–
Depots	22	**21**	**34.55**	**22.32**	3/1.74	11	104.55	121.24
Driver	20	**16**	28.21	2.76	3/1.31	14	**26.86**	**0.30**
Elevators	30	**27**	**55.00**	**13.90**	2/2.00	16	101.50	210.50
Ferry	50	50	**27.40**	**0.02**	2/1.98	50	32.88	0.03
Floortile	20	–	–		–	–	–	–
Freecell	20	**19**	**47.50**	**7.53**	2/1.62	17	62.88	68.25
Grid	5	**5**	**36.00**	**22.66**	3/2.12	3	195.67	320.65
Gripper	50	50	101.00	3.03	2/2.00	50	**99.04**	**0.36**
Logistics	28	28	**54.25**	2.61	2/2.00	28	56.25	**0.33**
Miconic	50	50	**42.44**	0.08	2/2.00	50	42.72	**0.01**
Mprime	35	27	**6.65**	**84.80**	2/2.00	28	17.92	204.76
Mystery	30	27	**6.47**	42.89	2/1.19	28	7.60	**15.44**
NoMystery	20	–		–	–	**6**	–	–
OpenStacks	30	**13**	**105.23**	**0.53**	3/1.80	7	112.42	6.49
OpenStacksIPC6	30	26	**29.43**	108.27	4/1.48	**30**	32.14	**23.86**
ParcPrinter	30	9	16.00	0.06	3/1.28	**30**	**15.67**	**0.01**
Parking	20	17	**39.50**	**38.84**	2/1.14	2	68.00	686.72
Pegsol	30	6	**16.00**	1.71	4/1.09	**30**	16.17	**0.06**
Pipes-NonTan	50	**45**	**26.36**	**3.23**	3/1.62	25	113.84	68.42
Pipes-Tan	50	**35**	**26.00**	205.21	3/1.63	14	33.57	**134.21**
PSRsmall	50	25	**13.79**	28.37	4/2.27	**44**	18.04	**4.99**
Rovers	40	**27**	**38.47**	**108.59**	2/1.39	20	67.63	148.34
Satellite	20	19	38.63	216.69	2/1.29	20	**34.11**	**8.44**
Scanalyzer	30	26	**26.81**	**33.96**	2/1.16	28	28.50	129.42
Sokoban	30	3	**80.67**	**7.83**	3/2.58	**23**	166.67	14.30
Storage	30	**25**	**12.62**	**0.06**	2/1.48	16	29.56	8.52
Tidybot	20	7	**42.00**	532.27	3/1.81	**16**	70.29	**184.77**
Tpp	30	24	**82.95**	**68.32**	3/2.03	23	116.45	199.51
Transport	30	**21**	**54.53**	94.61	2/2.00	17	70.82	**70.05**
Trucks	30	2	**31.00**	**4.58**	2/2.00	**8**	34.50	14.08
Visitall	20	**19**	**199.00**	**0.91**	1/1.00	3	2485.00	174.87
Woodworking	30	**30**	**21.50**	**6.26**	2/1.07	12	42.50	81.02
Zeno	20	19	**34.89**	166.84	2/1.83	20	35.11	**101.06**
Summary	1150	819	**44.4**	**55.01**	2.5/1.6	789	137.0	91.05

Table 3: Blind-Search *SIW* vs. Heuristic GBFS with h_{add} over real benchmarks (with joint goals). *I* is number of instances, *S* is number of solved instances, *Q* is average plan length, *T* is average time in seconds. *M/A* w_e stand for max and avg effective width per domain. *T* and *Q* reported for problems solved by both planners. Shown in bold are the numbers *S*, *Q*, or *T* that one planner improves over the other by more than 10%.

$|G'| = k$. It then sets G_k to G'. This is how *SIW* manages to use *IW* for both constructing the serialization and solving the subproblems.

The *SIW* algorithm is sound and the solution to P can be obtained by concatenating the solutions to the problems P_1, ..., P_m, where $m = |G|$. Like *IW*, however, *SIW* does not guarantee optimality. Likewise, while the *IW* algorithm is complete, *SIW* is not. The reason is that the subgoal mechanism implicit in *SIW* commits to intermediate states from which the goal may not be reachable. Of course, if there are no dead-ends in the problem, *SIW* is complete.

We have compared experimentally the blind-search algorithm *SIW* to a baseline heuristic search planner using a Greedy Best First Search (GBFS) and the additive heuristic (Bonet and Geffner 2001). Neither planner is state-of-the-art, as neither uses key techniques such as helpful actions or landmarks (Hoffmann and Nebel 2001; Richter and Westphal 2010). Still, the comparison shows that the non-goal oriented form of pruning in *IW* and the simple form of decomposition in *SIW* are quite powerful; as powerful indeed, as the best heuristic estimators.

SIW and GBFS are both written in C++ and use Metric-FF as an *ADL* to *Propositional STRIPS* compiler (Hoffmann 2003). The experiments were conducted on a dual-processor running at 2.33 GHz and 2 GB of RAM. Processes time or memory out after 30 minutes or 2 GB. The results are summarized in Table 3 . Out of 1150 instances, *SIW*solves 30 problems more, it's usually faster, and produces shorter solutions.

Table 3 shows also the highest and average effective widths of the subproblems that result from the serializations generated by *SIW*. The maximal effective width is 4, which

occurs in two domains: *8puzzle* and *PSRsmall*. On average, however, the effective width is between 1 and 2, except for four domains with effective widths between 2 and 3: Sokoban (2.58), PSRsmall (2.27), Grid (2.12), and TPP (2.03).

State-of-the-art Performance

While the blind-search *SIW* procedure competes well with a greedy best-first search planner using the additive heuristic, neither planner is state-of-the-art. Since state-of-the-art performance is important in classical planning, we show next that it is possible to deliver such performance by integrating the idea of novelty that arises from width considerations, with known techniques such as helpful actions, landmarks, and heuristics. For this we switch to a *plain forward-search best-first search planner* guided by an evaluation function $f(n)$ over the nodes n given by

$$f(n) = novelha(n) \quad (1)$$

where $novelha(n)$ is a measure that combines novelty and helpful actions, as defined below. In addition, ties are broken lexicographically by two other measures: first, $usg(n)$, that counts the number of subgoals not yet achieved up to n, and second, $h_{add}(n)$, that is the additive heuristic.

The subgoals are the problem landmarks (Hoffmann, Porteous, and Sebastia 2004) derived using a standard polynomial algorithm over the delete-relaxation (Zhu and Givan 2003; Keyder, Richter, and Helmert 2010). The count $usg(n)$ is similar to the landmark heuristic in LAMA (Richter, Helmert, and Westphal 2008), simplified a bit: we use only atomic landmarks (no disjunctions), sound orderings, and count a top goal p as achieved when goals q that must be established before q have been achieved (Lipovetzky and Geffner 2011).

The $novelha(n)$ measure combines the novelty of n and whether the action leading to n is helpful or not (Hoffmann and Nebel 2001). The novelty of n is defined as the size of the smallest tuple t of atoms that is true in n and false in *all previously generated nodes* n' *in the search with the same number of unachieved goals* $usg(n') = usg(n)$. Basically, nodes n and n' in the search with different number of unachieved goals, $usg(n) \neq usg(n')$, are treated as being about different subproblems, and are not compared for determining their novelty. The novelty of a node $novel(n)$ is computed approximately, being set to 3 when it's neither 1 nor 2. Similarly, if $help(n)$ is set to 1 or 2 according to whether the action leading to n was helpful or not, then $novelha(n)$ is set to a number between 1 and 6 defined as

$$novelha(n) = 2[novel(n) - 1] + help(n)\,. \quad (2)$$

That is, $novelha(n)$ is 1 if the novelty of n is 1 and the action leading to n is helpful, 2 if the novelty is 1 and the action is not helpful, 3 if the novelty is 2 and the action is helpful, and so on. Basically, novel states (lower $novel(n)$ measure) are preferred to less novel states, and helpful actions are preferred to non-helpful, with the former criterion carrying more weight. Once again, the criterion is simple and follows from performance considerations.

We call the resulting best-first search planner, BFS(f), and compare it with three state-of-the-art planners: FF, LAMA, and PROBE (Hoffmann and Nebel 2001; Richter, Helmert, and Westphal 2008; Lipovetzky and Geffner 2011).[2] Like LAMA, BFS(f) uses delayed evaluation, a technique that is useful for problems with large branching factors (Richter and Helmert 2009).

Table 4 compares the four planners over the 1150 instances. In terms of coverage, BFS(f) solves 5 more problems than LAMA, 18 more than PROBE and 161 more than FF. Significant differences in coverage occur in *Sokoban*, *Parking*, *NoMystery* and *Floortile* where either LAMA or BFS(f) solve 10% more instances than the second best planner. The largest difference is in *NoMystery* where BFS(f) solves 19 instances while LAMA solves 11.

Time and plan quality averages are computed over the instances that are solved by BFS(f), LAMA and PROBE. FF is excluded from these averages because of the large gap in coverage. LAMA and PROBE are the fastest in 16 domains each, and BFS(f) in 5. On the other hand, BFS(f) finds shorter plans in 15 domains, PROBE in 13, and LAMA in 10. The largest differences between BFS(f) and the other planners are in *8puzzle*, *Parking*, *Pipesworld Tankage*, *Pipesworld Non Tankage*, and *VisitAll*.

The results show that the performance of BFS(f) is at the level of the best planners. The question that we address next is what's the contribution of the four different ideas combined in the evaluation function $f(n)$ and in the tie-breakers; namely, the additive heuristic $h_{add}(n)$, the landmark count $usg(n)$, the novelty measure $novel(n)$, and the helpful action distinction $help(n)$. The last two terms are the ones that determine the evaluation function $f(n)$ in (1) through the formula (2).

Table 5 shows the result of a simple ablation study. The first row shows the results for the planner BFS(f) as described above, while the following rows show results for the same planner with one or several features removed: first delayed evaluation, then the additive heuristic, helpful actions, and novelty. This is achieved by setting $help(n) = 0$, $h_{add}(n) = 0$, and $novel(n) = 1$ respectively in (1) and (2) for all n. As it can be seen from the table, the greatest drop in performance arises when the novelty term is dropped. In other words, the novelty measure is no less important in the BFS(f) planner than either the helpful action distinction or the heuristic. The most important term of all however is the $usg(n)$ that counts the number of unachieved goals, and whose effect is to 'serialize' the best-first search to the goal without giving up completeness (as *SIW*). Moreover, the definition of the novelty measure in (2) uses the $usg(n)$ count to delimit the set of previously generated states that are considered. Yet BFS(f') with the evaluation function $f' = usg(n)$, i.e., without *any* of the other features, solves just 776 instances. On the other hand, with the term $novelha(n)$ added, the number jumps to 965, surpassing FF that solves 909 problems. This is interesting as BFS(f') uses no heuristic estimator then.

[2] FF is FF2.3, while PROBE and LAMA are from the 2011 IPC.

		BFS(f)			PROBE			LAMA'11			FF		
Domain	I	S	Q	T	S	Q	T	S	Q	T	S	Q	T
8puzzle	50	50	**45.45**	0.20	50	61.45	**0.09**	49	58.24	0.18	49	52.61	0.03
Barman	20	20	174.45	281.28	20	169.30	12.93	20	203.85	**8.39**	–	–	–
Blocks World	50	50	54.24	2.40	50	**43.88**	**0.23**	50	88.92	0.41	44	39.36	66.67
Cybersecure	30	28	39.23	70.14	24	52.85	69.22	30	37.54	576.69	4	29.50	0.73
Depots	22	22	49.68	56.93	22	44.95	**5.46**	21	61.95	46.66	22	51.82	32.72
Driver	20	18	48.06	57.37	20	60.17	1.05	20	46.22	**0.94**	16	25.00	14.52
Elevators	30	30	129.13	93.88	30	107.97	26.66	30	**96.40**	**4.69**	30	85.73	1.00
Ferry	50	50	32.94	0.03	50	29.34	**0.02**	50	28.18	0.31	50	27.68	0.02
Floortile	20	**7**	43.50	**29.52**	5	45.25	71.33	5	49.75	95.54	5	44.20	134.29
Freecell	20	20	64.39	**13.00**	20	62.44	41.26	19	68.94	27.34	20	64.00	22.95
Grid	5	5	70.60	7.70	5	**58.00**	9.64	5	70.60	**4.84**	5	61.00	0.27
Gripper	50	50	101.00	0.37	50	101.00	**0.06**	50	**76.00**	0.36	50	76.00	0.03
Logistics	28	28	56.71	0.12	28	55.36	**0.09**	28	**43.32**	0.35	28	41.43	0.03
Miconic	50	50	34.46	0.01	50	44.80	**0.01**	50	**30.84**	0.28	50	30.38	0.03
Mprime	35	35	10.74	19.75	35	14.37	28.72	35	**9.09**	**10.98**	34	9.53	14.82
Mystery	30	27	7.07	**0.92**	25	7.71	1.08	22	7.29	1.70	18	6.61	0.24
NoMystery	20	**19**	24.33	**1.09**	5	25.17	5.47	11	24.67	2.66	4	19.75	0.23
OpenStacks	30	29	141.40	129.05	30	137.90	64.55	30	142.93	**3.49**	30	155.67	6.86
OpenStacksIPC6	30	30	125.89	40.19	30	134.14	48.89	30	130.18	**4.91**	30	136.17	0.38
ParcPrinter	30	27	35.92	6.48	28	36.40	0.26	30	37.72	0.28	30	42.73	0.06
Parking	20	17	90.46	577.30	17	146.08	693.12	**19**	87.23	**363.89**	3	88.33	945.86
Pegsol	30	30	24.20	**1.17**	30	25.17	8.60	30	25.90	2.76	30	25.50	7.61
Pipes-NonTan	50	47	**39.09**	35.97	45	46.73	**3.18**	44	57.59	11.10	35	34.34	12.77
Pipes-Tan	50	40	**40.48**	254.62	43	55.40	102.29	41	48.60	**58.44**	20	31.45	87.96
PSRsmall	50	48	22.15	2.62	50	21.40	**0.08**	50	**18.31**	0.36	42	16.71	63.05
Rovers	40	40	105.08	44.19	40	109.97	24.19	40	108.28	**17.90**	40	100.47	31.78
Satellite	20	20	36.05	1.26	20	37.05	0.84	20	40.75	0.78	20	37.75	0.10
Scanalyzer	30	27	29.37	7.40	28	25.15	**5.59**	28	27.52	8.14	30	31.87	70.74
Sokoban	30	23	220.57	125.12	25	233.48	39.63	**28**	213.00	58.24	26	213.38	26.61
Storage	30	20	20.94	4.34	21	**14.56**	**0.07**	18	24.33	8.15	18	16.28	39.17
Tidybot	20	18	62.94	198.22	**19**	**53.50**	35.33	16	62.31	113.00	15	63.20	9.78
Tpp	30	30	112.33	36.51	30	155.63	58.98	30	119.13	**18.12**	28	122.29	53.23
Transport	30	30	107.70	55.04	30	137.17	44.72	30	108.03	94.11	29	117.41	167.10
Trucks	30	15	26.50	8.59	8	26.75	113.54	16	24.12	**0.53**	11	27.09	3.84
Visitall	20	20	**947.67**	84.67	19	1185.67	308.42	20	1285.56	77.80	6	450.67	38.22
Woodworking	30	30	41.13	19.12	30	41.13	15.93	30	51.57	**12.45**	17	32.35	0.22
Zeno	20	20	37.70	77.56	20	44.90	6.18	20	35.80	**4.28**	20	30.60	0.17
Summary	1150	1070	87.93	63.36	1052	98.71	49.94	1065	98.67	44.35	909	67.75	51.50

Table 4: BFS(f) vs. LAMA, FF, and PROBE. *I* is number of instances, *S* is number of solved instances, *Q* is average plan length, *T* is average time in seconds. *T* and *Q* averages computed over problems solved by all planners except FF, excluded because of the large gap in coverage. Numbers in bold show performance that is at least 10% better than the other planners.

Discussion

We have introduced a width parameter that bounds the complexity of classical planning problems along with an iterative pruned breadth-first algorithm *IW* that runs in time exponential in the problem width. While most benchmark domains appear to have a bounded and small width provided that goals are restricted to single atoms, they have large widths for arbitrary joint goals. We have shown nonetheless that the algorithm derived for exploiting the structure of planning problems with low width, *IW*, also pays off over benchmarks with joint goals once the same algorithm is used for decomposing the problems into subproblems. Actually, the resulting blind-search algorithm *SIW* is competitive with a baseline planner based on a Greedy Best First Search and the additive heuristic, suggesting that the two ideas underlying *SIW*, novelty-based pruning and goal decomposition, are quite powerful. We have also shown that it is possible to integrate the notion of novelty derived from width considerations, with existing planning techniques for defining the evaluation function of a novel best-first search planner with state-of-the-art performance. Moreover, we have shown that the new technique contributes to the performance of this planner no less than helpful actions or heuristic estimators.

The notion of width is defined over graphs $\mathcal{G}^m$ whose vertices are tuples of at most m atoms. This suggests a relation between the width of a problem and the family of admissible h^m heuristics which are also defined over tuples of at most m atoms (Haslum and Geffner 2000). A conjecture that we considered is whether a width of w implies that h^w is equal to the optimal heuristic h^*. The conjecture however is false.[3] It turns out that for this correspondence to be true,

[3] A counterexample is due to Blai Bonet. Consider a problem P

	I	S	% S	Q	T
BFS(f)	1150	1070	93%	82.80	62.89
No Delayed Eval	1150	1020	89%	80.67	65.92
No Heuristic	1150	965	84%	100.47	32.43
No Helpful Actions	1150	964	84%	81.82	64.20
No Novelty	1150	902	78%	86.40	46.11

Table 5: Ablation study of BFS(f) when some features are excluded. Delayed evaluation excluded from second and following rows, in addition to feature shown. Columns show number of instances (I), number of instances solved (S), % solved (%S), and average plan lengths (Q) and times in seconds (T).

an additional clause is needed in the definition of the heuristic h^m; namely, that $h^m(t)$ is no lower than $h^m(t')$ when t' is a tuple of at most m atoms that implies t in the sense defined in Section 3. Yet, checking this implication in general is intractable.

In this paper, we have only considered planning problems where actions have uniform costs. Some of the notions and algorithms developed in the paper, however, extend naturally to non-uniform costs provided that the breadth-first search in *IW* is replaced by a uniform-cost search (Dijkstra).

Acknowledgments. H. Geffner is partially supported by grants TIN2009-10232, MICINN, Spain, and EC-7PM-SpaceBook.

References

Amir, E., and Engelhardt, B. 2003. Factored planning. In *Proc. IJCAI-03*.

Bäckström, C. 1996. *Five years of tractable planning*. IOS Press. 19–33.

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

Brafman, R. I., and Domshlak, C. 2006. Factored planning: How, when, and when not. In *Proc. AAAI-06*.

Bylander, T. 1994. The computational complexity of STRIPS planning. *Artificial Intelligence* 69:165–204.

Chen, H., and Giménez, O. 2007. Act local, think global: Width notions for tractable planning. In *Proc. ICAPS-07*.

Dechter, R. 2003. *Constraint Processing*. Morgan Kaufmann.

Freuder, E. 1982. A sufficient condition for backtrack-free search. *J. ACM* 29:24–32.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. AIPS-00*.

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

with initial situation $I = \{r\}$, goal $G = \{z\}$, and actions $a : r \rightarrow p, \neg r$, $b : r \rightarrow q, \neg r$, $c : r \rightarrow x$, $d : x \rightarrow p, q, y$, and $e : p, q \rightarrow z$. It can be shown that the problem has width $w = 1$, as the graph $\mathcal{G}^1$ contains the rooted path r, x, y, z. Yet, $h^*(P) = 3$ with optimal plan c, d, e, while the heuristic h^m for $m = 1$, is h_{max}, which for this problem is 2. Thus, $w(P) = w$ and yet $h^*(P) \neq h^w(P)$.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *JAIR* 22:215–278.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *JAIR* 20:291–341.

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

Hoffmann, J. 2011. Analyzing search topology without running any search: On the connection between causal graphs and h^+. *JAIR* 41:155–229.

Keyder, E.; Richter, S.; and Helmert, M. 2010. Sound and complete landmarks for and/or graphs. In *Proc. ECAI-10*.

Lipovetzky, N., and Geffner, H. 2011. Searching for plans with carefully designed probes. In *Proc. ICAPS-11*.

Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In *Proc. ICAPS-09*.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *JAIR* 39:122–177.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *Proc. AAAI-08*.

Zhu, L., and Givan, R. 2003. Landmark extraction via planning graph propagation. In *Proc. ICAPS-03 Doctoral Consortium*.

Probabilistically Reusing Plans in Deterministic Planning

DANIEL BORRAJO
Universidad Carlos III de Madrid
Avda. de la Universidad, 30. 28911 Leganés (Madrid), Spain
dborrajo@ia.uc3m.es

MANUELA VELOSO
Computer Science Department, Carnegie Mellon University, USA
Pittsburgh PA 15213
veloso@cs.cmu.edu

Abstract

Inspired by probabilistic path planning, we contribute a planning approach that probabilistically balances heuristics and past plans as guidance to planning search. Our ERRT-PLAN algorithm generates multiple search branches probabilistically choosing to extend them towards the current goal or towards actions or goals of a past given plan. We have defined domains to show where current techniques could be trapped. Also, we show experimental results with a variety of domains, where we show the strengths of ERRT-PLAN.

Introduction and Related Work

As the complexity of planning is realized, researchers create a variety of planning approaches to try to increase the scalability horizon of their planning search. We view three different classes of domain-independent approaches to automated planning, as we sketch in Figure 1.[1] Given a domain D, and a new problem P, planners can extract domain-independent heuristics to guide their search. Such "Planning from scratch" has shown to be very efficient in large classes of problems, as is the case of forward-chaining search using a combination of a well-informed heuristic, coming from a relaxed planning graph (Hoffmann & Nebel 2001). "Planning with learning" can in addition rely on training experience as compiled into domain-dependent heuristics. And "Planning with reuse" can use specific past solution plans.

In this work, we focus on planning with reuse, as we assume that a similar past solution plan may be available in several situations where the new planning problem naturally just deviates from a past problem, as for example may be the case when there is a need for replanning during plan execution.

Thus, according to the three classes of domain-independent guidance defined in Figure 1, the closest related works to the one presented in this paper are reuse approaches. They could

[1]We leave out from this classification all domain-dependent planning approaches, as Hierarchical Task Networks (Nau *et al.* 2003) or approaches based on manually defined domain-dependent heuristics (Bacchus & Kabanza 2000).

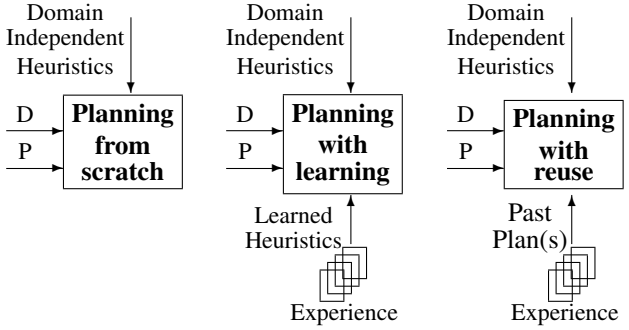


Figure 1: A view of planning according to search guidance.

be classified in turn as single plan reuse and multiple plans reuse. In the first category, we find all techniques that reuse only one plan, and this is the category where our work lies in. It mainly focuses on replanning and uses the current plan in execution to guide the search for the new plan. So, when the execution of a plan fails, the planner is invoked to generate a new plan from the current state and plan and the goals. As an example, LPG-ADAPT (Fox *et al.* 2006) focuses on replanning with minimum changes to previous plans using the LPG planner. They adapted LPG plan-modification heuristics, trying to maximize the reused actions in the new plan. We have in common that both approaches rely on stochastic search, and in reusing the plan from a previous search. However, their goal is to maximize the reused actions (plan stability), while ours is guiding the new search by using previous plans, not necessarily requiring plan stability. Thus, as we show in the experiments, they find problems even if the previous problem was very similar to the new one. We also add goal-reuse, which is not considered by them.

In the second category, multiple plans reuse, we find case-based reasoning applied to planning or case-based planning approaches (Veloso & Carbonell 1993; Kambhampati & Hendler 1992; Serina 2010; De la Rosa, García-Olaya, & Borrajo 2007). But, we focus only on the reuse of one solution, as the approaches in the previous paragraph. So, we do not deal here with other CBR tasks as the computation of a similarity metric, or the efficient storage of past cases, and any reasonable previous approach would work well with our approach.

At this time, we therefore assume that a past plan is given that is found to be similar to the new problem, and address the question on how the planner should reuse the past solution plan in its new search for a solution to the new problem. All of the past case-based planning or planning with reuse approaches, to our knowledge, proceed by trying to adapt the past plan with the hope that such adaptation will lead to improved search time, and with no significant sacrifice to plan quality. However, given the unpredicted effect of the fact that the past planning problem is only *similar* to the new problem, the adaptation process is known not to be guaranteed neither to save planning effort nor to hold plan quality (Nebel & Koehler 1995). But it is also the case that being guided by a past solution plan has the potential (not the guarantee) to be beneficial. Our planning with reuse algorithm differs from past algorithms in the fact that it considers both the potential "danger" and "benefit" of using a *similar* past plan.

Concretely, we contribute in this paper a *probabilistic* planning with reuse approach inspired by the ERRT algorithm (Bruce & Veloso 2002).[2] Our ERRT-PLAN algorithm probabilistically balances an heuristic-based guided search for a solution to the new problem and a past-plan guided search. The algorithm generates multiple search branches probabilistically choosing to extend them towards the current goal or towards actions or goals of a past given plan. Basically, ERRT-PLAN expands its search probabilistically as: heuristic search towards the goal of the current problem; reuse of an action of the past plan; and heuristic search towards a goal of the past plan. Through this probabilistic choice, ERRT-PLAN considers the past plan as a *bias* to its search for a new problem, and it does not try to deterministically adapt it. The past plan is hence present in the search as waypoints to direct the new search. As such, interestingly ERRT-PLAN can probabilistically both be well guided and not misguided by the past plan. There have been other approaches in planning that are based on the "Planning from Scratch" paradigm and RRT (Burfoot, Pineau, & Dudek 2006; Alcázar, Veloso, & Borrajo 2011). However, they belong to a different category as we do "Planning with Reuse".

The paper is organized as follows. First, ERRT is presented. We then present the ERRT-PLAN planning algorithm and show examples and results in two especially designed domains to show its advantages with respect to heuristic forward search from scratch. We then focus on conceptually comparing with examples our ERRT-PLAN with LPG-Adapt (Fox *et al.* 2006), as an available planner with reuse.[3] We also include some results using domains of the IPC. We draw conclusions summarizing the contributions of the work, while discussing the general aspect of our *probabilistic commitment* as not specific to our approach but of potential use to any other current planning technique.

[2]The RRT (Rapidly exploring Random Trees) algorithm (LaValle & Kuffner 2001) efficiently searches by sampling its space. ERRT (Execution extended RRT) (Bruce & Veloso 2002) further successfully includes past experience in the sampled search.

[3]We thank the authors of (Fox *et al.* 2006) for making the planner available to us and informing us on how to run it appropriately.

Path Planning. Extended RRT

We introduce the basic components on top of which we build our solution: the ERRT technique and the new search method. The heuristic planner we use is a re-implementation of the METRIC-FF planner (Hoffmann 2003). It uses forward-chaining search with the relaxed planning graph heuristic, and has several implemented search algorithms. As FF, our ERRT-PLAN search algorithm is built on top of EHC followed by a call to A* in case EHC fails. We use this scheme given that these techniques (heuristic and forward search techniques) are a backbone of most current planners.

Extended RRT (ERRT) is an extension over the standard RRT algorithm (LaValle & Kuffner 2001) targeted at efficient replanning. ERRT stores waypoints of a path, i.e., states in a path, and probabilistically reuses them in the next search (Bruce & Veloso 2002). ERRT takes as input an initial state, a goal state, a set of waypoints (states that were part of the solution path from a previous search) and two probabilities: p is the probability that at a given search step ERRT will focus on the goal state; and r is the probability with which at a given search step ERRT will focus on a previous waypoint (instead of the goal state). Finally, with a probability of $1-p-r$, it will focus towards a random state. The random target ensures a level of exploration of potential benefit in path planning with obstacles. Note that the three sources (goal, past plan, and random) act as "biases" to the new ERRT search. Through the adjustment of the different probabilities, the algorithm gives preference to one or another bias, e.g., if the environment and the goal are of low dynamics, then a high value of r is desirable, as the new search may be quite similar to the past one. In contrast, if the dynamics are high, a high value of p captures a preference to a new search rather than plan reuse.

ERRT-PLAN

We now present our ERRT-PLAN algorithm, inspired by ERRT, but now suitable for classical task planning problems. We can use this algorithm with any search technique, but in this paper we have used it in combination with EHC. We will first define the main concepts.

Definition1. A planning problem is a tuple $< P, A, I, G >$, where P is a set of propositions, A is a set of grounded actions, $I \subseteq P$ is the initial state and $G \subseteq P$ is the set of goals. As in STRIPS planning, each action $a \in A$ is represented as three sets: $pre(a) \subseteq P$, $add(a) \subseteq P$ and $del(a) \subseteq P$ (preconditions, adds and deletes).

Definition2. A plan $\pi = \{a_1, a_2, \ldots, a_n\}$ is an ordered set of grounded actions $a_i \in A$, that achieves a goal state from the initial state I ($G \subseteq \delta(a_n, \delta(a_{n-1}, \ldots \delta(a_1, I) \ldots))$).[4]

Definition3. Given a plan π, the set of weakest preconditions, wp_i, of any given action $a_i \in \pi$ is the set of propositions that are required to be true in the state before applying a_i, so that the goals can be achieved from a_i by applying the actions in the remaining of the plan $\{a_{i+1}, \ldots, a_n\}$.

The standard way of computing this set for each action in the plan consists of performing goal regression on the totally-ordered plan. Since there might be parallel paths to achieve the goals, we use a slightly different version of weakest preconditions. First, we compute a partially-ordered plan from the totally-ordered plan returned by most current planners. Then, we compute the weakest preconditions of each action on the remaining of the branch of the partially-ordered plan from that action towards the goal. In the partially-ordered plan, two ficticious actions are inserted at the beginning, a_o, and at the end of the plan, a_∞. The weakest preconditions are usually computed by goal regression as: $wp_i = pre(a_i) \cup \bigcup_{a_j \in T(a_i)} pre(a_j) \cap s_i$. $pre(a_i)$ are the preconditions of action a_i, $T(a_i)$ is the set of actions in the transitive closure of causal links from a_i to the end action a_∞ (all actions that come after a_i in the same branch of the partial plan), and s_i is the state before applying a_i. One last improvement is that we remove from that set those literals that are static (no action adds or deletes them). As we will see next, we will use the weakest preconditions of each action in the plan as potential subgoals to focus on during search.

The Algorithm

The algorithm is an adaptation of ERRT. Figure 2[5] shows the top-level ERRT-PLAN algorithm, which takes as input the planning problem (i.e., the initial state I and set of goals G), the domain description (i.e., the set of actions A), and our new ERRT-PLAN specific probabilities for node expansion: p towards the goal; r_a (action-reuse) towards an action of the past plan; and $r_g = 1 - p - r_a$ (goal-reuse) towards a goal in the past plan, as we further explain below. A last input W (waypoints) contains the solution (plan) represented as an ordered set of pairs of actions and their weakest preconditions, a_i, wp_i:

$W = [(a_0, wp_0), (a_1, wp_1), \ldots, (a_k, wp_k)]$. Action-reuse targets some a_i while goal-reuse targets some wp_i.

There are two main differences between ERRT and ERRT-PLAN. First, in ERRT-PLAN, the distance between nodes is

[4]The function $\delta(a_i, S_j)$ returns the state after applying action a_i in state S_j.

[5]For the sake of clarity, we do not show the failure conditions, but they relate to dead-ends or reaching resource bounds (time or memory).

function ERRT-PLAN(I, G, A, p, r_a, W): plan

I: initial state
G: goal state
A: set of domain actions
p: probability of selecting EHC in the expansion of each node
r_a: probability of doing action-reuse
($r_g = 1 - (p + r_a)$: will be the probability of doing goal-reuse)
$W = [(a_0, wp_0), (a_1, wp_1), \ldots, (a_k, wp_k)]$:
array of actions a_i and their weakest preconditions wp_i

---

tree = $\{I\}$
while not(**CloseEnough**(tree,G)) do
i: UniformRandom in [0.0 .. 1.0]
if $0 <$ i $< p$
then tree = expand-tree(EHC,I, G, A, ϕ,tree)
else if $p <$ i $< p + r_a$
then tree = expand-tree(**action-reuse**,I, G, A, W,tree)
else tree = expand-tree(**goal-reuse**,I, G, A, W,tree)
return **Extract-plan**(tree)

Figure 2: The ERRT-PLAN algorithm.

computed according to the planning heuristic. The function **CloseEnough** checks whether the goal is true in the heuristically closest node of the tree. The second difference relates to the representation of states and goals. While in ERRT both states and goals are represented as vectors of N dimensions, in planning, initial states and goals are represented as sets of literals. Furthermore, goals in task planning are partial descriptions of states, so they cannot be used interchangeably with states as in ERRT.

ERRT-PLAN carries the planning search expanding its search tree using three possible biases, namely using EHC, action-reuse, or goal-reuse. The search tree keeps multiple open nodes (internal and leaves), and all such nodes are candidates for expansion. At each iteration, one of the nodes in the open list is found to be the closest to the chosen target and is expanded. The children of expanded nodes are not added to the search tree immediately. They remain as children until the search chooses to expand one of them and adds it to the tree. As in the ERRT algorithm, the expansion of the tree includes the three main functions to: choose a target (**ChooseTarget**); compute the closest node to the chosen target (**Nearest**); and extend such closest node (**Extend**). We now present these three functions for the EHC, action-reuse, and goal-reuse expansions.

- The EHC expansion corresponds to:
  - **ChooseTarget** selects the goal G as the current target;
  - **Nearest** returns the closest open node to the chosen target;
  - **Extend** iteratively generates the children of nearest, and as soon as one returns a heuristic value less than nearest, it stops and adds that node to the tree. If no

child is better than nearest, it performs a breadth-first search until it finds a better one. This is implemented similarly as in METRIC-FF by first expanding successors that are helpful actions (actions that are thought to be relevant to achieve the goals and can be directly applied in the current state).[6] If that fails, then we switch to non-helpful ones. As in (Burfoot, Pineau, & Dudek 2006), we set a limit on the number of nodes in this expansion. We start with a limit of 100 nodes, and then we increase it by 50, every time we perform a breadth-first search. Also, we add to the tree the path from nearest to the best node in that search (even if it is not better than the nearest heuristic value).

- The action-reuse expansion corresponds to:
  - **ChooseTarget** selects the goal G as the current target;
  - **Nearest** and **Extend** are done jointly. **Nearest** returns the first open node in which an action a_i from the past plan can be applied. In each node, we keep a pointer of the last action of the previous plan that was applicable in that node. We start searching for applicable actions at nodes whose pointer leads to actions closer to the end of the previous plan. If node n_i has a pointer to action a_k and node n_j has a pointer to action a_m, and $m > k$, then we first try to find an applicable action in node n_j starting at position $m+1$ of the previous plan. If an applicable action is found, then we try to apply as many actions as possible from the previous plan. We set the pointer to the position of the last applied action, so next time we start looking for applicable actions from that action on. When we create a node, its pointer is initialized to 0. And once we reach the end of the previous plan in a node, then that node is not tried again for action reuse, and we try the other nodes in the search tree. If no applicable action is found at any node, we do an EHC expansion. This allows solving some problems even in the case of using pure action reuse ($p = 0, r_a = 1.0, r_g = 0$) and when the previous solution does not solve completely the new problem.
- The goal-reuse expansion corresponds to:
  - **ChooseTarget** selects a goal from the previous search. It randomly selects the weakest preconditions of actions of the previous solution that have a later position than the one referred to by a global pointer, in the same spirit as the action-reuse (in this case we keep a global pointer instead of a node-dependent pointer). So, if the pointer is g, we randomly pick one weakest preconditions set from $< wp_{g+1}, \ldots, wp_k >$. If none is available ($g \geq k$), then it returns the goals of the current problem G.
  - **Nearest** computes the heuristic distance from each node in the search tree and the selected target, and returns the node with the lowest heuristic value. If more than one, then the deepest one is preferred. Given that the computation of the heuristic distance of each node in the tree towards a given goal (the problem goals or one of the weakest preconditions of the previous solution) can be repeated many times, we cache in a hash table the results of those heuristic computations.
  - **Extend** expands the nearest node, if not already expanded, and adds one of the successors according to EHC. So, this is exactly the same behavior as the **Extend** function of the EHC expansion.

[6] They are computed at the same time as the heuristic value of nodes.

Experiments

In this section we will show through several examples the benefits of the approach when dealing with tasks for which ERRT-PLAN outperforms other approaches. In particular, we compare against a baseline planner, FF, and a re-planning algorithm, LPG-ADAPT. We focus our study on the features of the tasks that make it particularly difficult for them. More specifically we will present examples for the following cases:

- heuristic search approaches might fail, and the previous plan can help on solving the task
- but, strongly relying on past plans might also not be always beneficial:
  - LPG-ADAPT does not remove unnecessary parts of past plans
  - some small difference in the initial state makes LPG-ADAPT generate non-optimal plans
  - small differences in the goals make LPG-ADAPT use parts of the previous plan, but there is a better solution to solve the difference if you do not use those parts
  - past plans are reused, while other parts are dynamically build by the planner

We have used 0.0, 0.3, 0.7 and 1.0 as parameters values for both p and r_a (resulting on the same range of values for r_g), removing those configurations with $p + r_a + r_g > 1.0$. Since ERRT-PLAN is stochastic, we run each ERRT-PLAN configuration five times and show the median of the results. The base configuration in most experiments was the standard FF algorithm (EHC followed by A* in case no solution was found) with the relaxed planning graph heuristic. It can also be seen as a special case of ERRT-PLAN, when $p = 1.0, r_a = 0.0, r_g = 0.0$. We used a time bound of 600s and a memory bound of 3Mb.[7]

On Solving Difficult Planning Tasks

As it is well known, no single technique is better than the others in all domains. For instance, one of the best domain-independent heuristics for planning, the relaxed plan heuris-

[7] No problem could not be solved due to exceeding the memory bound.

tic of FF that is used one way or another in most current planners, can lead a greedy search algorithm to dead-ends. Unfortunately, many real world tasks present dead-ends. Some planners, as FF or LAMA, try to avoid dead-ends by resorting to best-first search and/or combining it with other heuristics. But in some domains, even with a very good heuristic, best-first search can expand an arbitrarily large number of nodes (Helmert & Roger 2008). The knowledge on previous plans can help guide the planner to avoid those dead-ends or exploring a huge useless area of the state space. In this section, we show that ERRT-PLAN stochastically selects whether to reuse parts of the previous plans or to build some parts from scratch. This allows ERRT-PLAN to solve tasks with dead-ends and show how we can improve over the combination of the relaxed plan graph heuristic+EHC on those domains by using ERRT-PLAN.

We contribute a new planning domain, *NewGrid*, similar to the ones used by the path-planning community, where we define two new planning tasks. We show that these tasks are well targeted at illustrating the features of our technique. This domain is a proof-of-concept of the problems of the combination of EHC +planning graph heuristic.[8] In this domain, a robot has to reach any of the several positions that have gold on them. Gold is stored on closed boxes that can be opened by a key that the robot carries at the beginning. The world is defined as a grid with walls and obstacles. Walls can never be traversed. Obstacles can be traversed, but once they are traversed by the robot, the robot loses the key that it carries. Thus, once the robot traverses a cell with an obstacle, it cannot open any box, resulting in a dead-end (no goal is reachable from those states). Given that losing the key appears as a delete effect, the relaxed plan heuristic does not detect those situations. So, the combination of the relaxed plan heuristic and any non-backtraking search technique (as, for instance, EHC) leads to dead-ends. Obstacles can be thought in general as places where we lose our non-renewable resources, or highly dangerous areas in other domains. Figure 3 shows examples of scenarios of this domain with two goals (G) and one obstacle (O) (a), three goals and two obstacles (b), and one goal and several obstacles (c and d). The robot appears as R.

In this domain, and in some others also, given that EHC is a very greedy search algorithm, the order in which the nodes are studied is very relevant. This aspect influences the behavior of EHC-based planners. So, we created two types of tasks in the *NewGrid* domain to highlight the difficulties that we can find using it: the L (a and b) and Cup tasks (c and d). The names refer to the shape of the cells without obstacles. Figure 3 shows an example of training and test problems for both tasks. These tasks could not be solved by EHC because it tried to eagerly follow the shortest path to the goal. Thus, it does not notice that when traversing the obstacles it loses the key and cannot solve the problem. If we follow another

[8]Similar domains, as GoldMiner or Matching Blocksworld, can be found in the first learning track of the IPC held at ICAPS'08 (http://eecs.oregonstate.edu/ipc-learn/).

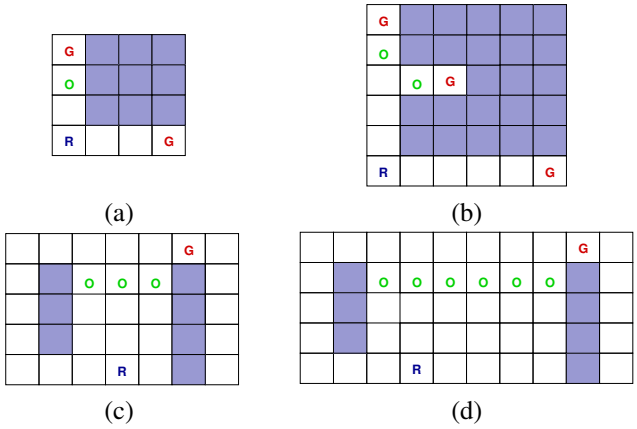


Figure 3: Example tasks in the L and Cup tasks in the *NewGrid* domain. (a) training L problem, (b) example of 6×6 L task with one corridor, (c) training Cup problem, and (d) Cup task with 10 columns. R is the robot, O represents an obstacle and G a goal.

algorithm after failing (as FF does using A*), it can take it a huge amount of time to find the solution, given that the heuristic continues to force the search algorithm to select nodes that are in the wrong direction. In the case of the L tasks (a) and (b), given that EHC evaluates the successors of a node in a given order, if it decides to move to the places where gold comes after passing though an obstacle, it will find dead-ends. And, if a planner switches to A* (as it is the case of FF), it might be that we have a high number of dead-end corridors branching out of that part of the search tree, so that it can take a long time to find the right branch. If, by chance, the order of the successors makes it to move to the right on (a) or (b), we can always create the complementary version in which all the corridors are on the bottom part and branching from it will lead to dead-ends.

In the second task, that we call the Cup task, the heuristic finds that going directly towards the only goal is better than going first to the left of the wall. So, EHC will never find a solution of this task. If we change to A* and the space among the walls is sufficiently large, it will have to explore all that area before going to the left of the wall. So, the time to solve the problem grows quadratically with the size of that area,[9] as shown in the following experiments.

For these tasks, we generated problems by increasing size of the grid from 10×10 to 70×70 in increments of 10 in the L task. We also added a number of corridors that would branch out of the corridor going up (the one with the obstacle) that would be $N - 4$ where N is the number of rows/columns. Those corridors also hold gold at their ends, but always with an obstacle before them. In the Cup task, we fixed the number of rows and set the number of columns from 10 to 150 (in steps of 10) with the same structure for the walls and obstacles, and the same relative positions for

[9]We assume full duplicate detection is performed.

the robot (always starts at (0,3)) and gold position (always at $(N-1,7)$). In these two tasks, we solved the training problems (a) and (c) with the reimplementation of FF (EHC followed by A* where EHC could not solve either task).

Then, using the solution to these two simple problems (Figure 3(a) and (c)), we tried to solve the test problems with FF and with the ERRT-PLAN configurations. Figure 4 shows the time employed to solve the Cup task. The time spent to solve the problems is in most configurations four/five times less than the one employed by FF, where EHC could not solve a single problem and it had to always resort to A*. One of the potential drawbacks of ERRT-PLAN with respect to EHC could be the extra nodes that it keeps in memory. In the case of these experiments, the number of nodes explored by ERRT-PLAN was linear to (even almost the same as) the number of steps in the solution. The reason why time grows non-linearly is that the heuristic computation (FF heuristic) grows non-linearly with the size of the grid. It goes from an average of 0.001s per evaluation with $N = 10$ to around 100s in the case of $N = 150$. We also changed the initial position of the robot (by moving it one and two positions to the right), and of the goal (one position to the right) and results were similar.

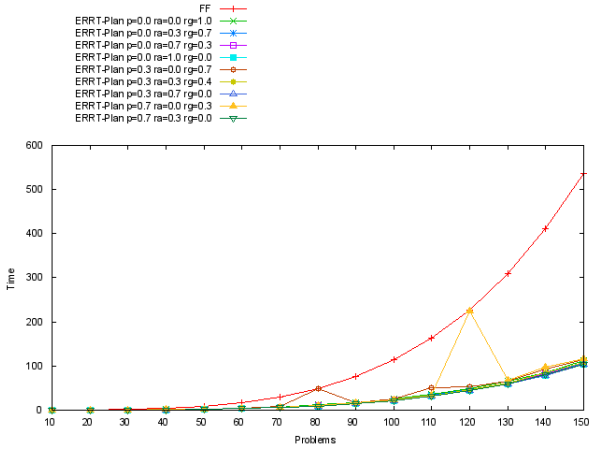


Figure 4: Time to solve problems on the *NewGrid* domain for the Cup task. ERRT-PLAN results are the median of planning time.

Figure 5 shows the time to solve problems in the L task. With respect to solvability, FF could not solve problems starting on $40-36$, even after resorting to A*. In the case of ERRT-PLAN, most configurations can solve problems of up to $70-56$ in the given time bound.

Strong Bias Towards Previous Plans Might Not Help

In the previous section, we have shown that guidance from the past plan can help planners to find solutions avoiding repeating the same mistakes as before. However, on the other

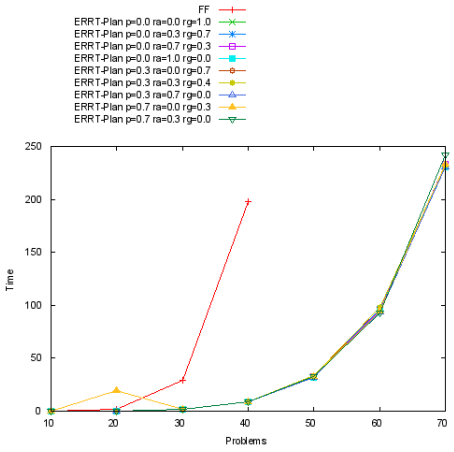


Figure 5: Time to solve problems on the *NewGrid* domain for the L task with maximum number of corridors (N rows/columns, $N-4$ corridors). ERRT-PLAN results are the median of planning time.

extreme, strongly relying on the past plan sometimes can also be a bad choice. In this section, we show that replanning systems as LPG-ADAPT find difficulties, because they are too tied to the previous plan. In fact, it starts the search in the plan graph space from the initial node representing the past plan. And it tries to diverge as little as possible from that plan by using stochastic local search with multiple restarts and an evaluation function that is biased towards maintaining stability (minimizing the difference of the new plan with respect to the previous plan). We show three examples where this scheme does not work well, because LPG-ADAPT returns bad quality solutions.

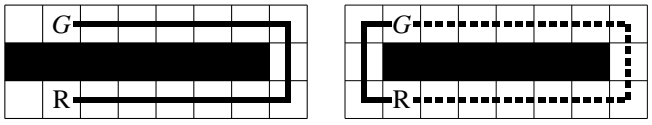
Avoiding Removing Past Actions. In the first example, it tries to avoid removing actions from the past plan, since it focuses on maximizing plan stability. So, sometimes the solutions contain unnecessary actions. Consider, for instance, the *Unnecessary-past-steps* domain on Figure 6. We first give LPG-ADAPT the solution to a previous problem that is only composed of action a1. If we now give it the problem with I={g2} and G={g+}, LPG-ADAPT generates the solution π={a1,a3,a1+}. In this solution, a1 is not needed, given that g+ is achieved with a1+, and its preconditions are g2 that is true in the initial state, and p3 that is achieved by a3. But, LPG-ADAPT tries to reuse as much as possible from the previous plan in order to solve the new problem. This is just a simple example of this behavior, but it is very easy to make the previous solution as long as we want and it still keeps the previous solution there. LPG-ADAPT is reluctant to remove that part of the previous plan, since it adds g2 that is needed for applying a1+, even if in the new problem this plan is not needed because g2 is true in the initial state. On the contrary, all configurations of ERRT-PLAN are able to find the right plan, getting rid of the unnecessary actions.

a1	**a1+**	**a2+**
Pre:	*Pre:* g2,p3	*Pre:* p4,p5
Adds: g1,g2	*Adds:* g+	*Adds:* g+
Dels:	*Dels:*	*Dels:*

a3	**a4**	**a5**
Pre:	*Pre:*	*Pre:*
Adds: p3	*Adds:* p4	*Adds:* p5
Dels:	*Dels:*	*Dels:*

Figure 6: *Unnecessary-past-steps* domain that shows how LPG-ADAPT generates plans with unnecessary actions.

Small Difference on Initial States. In the second example, a small variation in the initial state makes the previous plan highly suboptimal, but LPG-ADAPT returns the same previous plan, without noticing. Consider for instance two new simple tasks in the *NewGrid* domain graphically shown in Figure 7. The robot should go from its initial position, R, to the goal position, G. Suppose we use the plan for the problem on the left to bias the solution of the problem in the right where suddenly a new direct path appears – the obstacle in (0,1) disappears. LPG-ADAPT always selects the previous path to solve the problem. If asked for multiple solutions, then it finds the new path, but it returns suboptimal solutions (by repeating states, for instance). In the case of ERRT-PLAN, it consistently finds the new optimal path. Note that the corridor to the right might be as long as we want, and LPG-ADAPT behavior always selects that plan, instead of the new one. Also, one might call LPG-ADAPT for multiple solutions, but then we have created an arbitrary number of new holes in the wall going to the right, and it takes LPG-ADAPT a long time until discovering the optimal solution.



(a) (b)

Figure 7: (a) is the past problem and its solution and (b) is the same problem, except for a small difference in the initial state (the path in the left is not blocked). In (b), LPG-ADAPT solution is represented by the dashed line and ERRT-PLAN solution is shown by the solid line.

Small Difference on Goals. In the third example, a small variation in the goals makes LPG-ADAPT reuse past of the previous plan, without noticing there is a better solution. We base our example in the domains presented in (Veloso & Blythe 1994). More specifically, we merge the *Laboriously linkable* domain with our previous *Unnecessary-past-steps* domain. The *Linkability-Hidden* domain is shown in Figure 8. Suppose the solution to the previous problem is again $\pi = \{a1\}$ and the new problem is $I=\{\}$ and $G=\{g+\}$. LPG-ADAPT starts with a plan graph that contains a1 and tries to build a solution from that point on. The only actions that add g+, the only goal now, are a1+ and a2+. The evaluation function of LPG-ADAPT thinks a1+ is a better option since g2 is already true (through a1), g* can be achieved in one step with a*, and it thinks that p3 can be achieved in three steps (ap1, ap2 and ap3). On the other side, a2+ seems a worse option since it needs one step for achieving g3, another for g* and three steps to achieve q3 (aq1, aq2 and aq3). So, it selects a1+, which is the unoptimal solution, as shown in Figure 9.

a1	**a1+**	**a2+**	**a3**
Pre:	*Pre:* g2,p3,g*	*Pre:* g3,q3,g*	*Pre:*
Adds: g1,g2	*Adds:* g+	*Adds:* g+	*Adds:* g3
Dels:	*Dels:*	*Dels:*	*Dels:*

a*	**ap1**	**ap2**	**ap3**
Pre:	*Pre:* g*	*Pre:* p1,g*	*Pre:* p2,g*
Adds: g*	*Adds:* p1	*Adds:* p2	*Adds:* p3
Dels:	*Dels:* g*	*Dels:* p1,g*	*Dels:* p2,g*

aq1	**aq2**	**aq3**
Pre:	*Pre:* q1	*Pre:* q2
Adds: q1	*Adds:* q2	*Adds:* q3
Dels:	*Dels:* q1	*Dels:* q2

Figure 8: *Linkability-Hidden* domain to show how LPG-ADAPT generates suboptimal plans with small variations of the goals.

0.0000: (A1) [D:1.00; C:1.00]	
0.0000: (A*) [D:1.00; C:1.00]	
1.0000: (AP1) [D:1.00; C:1.00]	0: (A3) [1]
2.0000: (A*) [D:1.00; C:1.00]	1: (AQ1) [1]
3.0000: (AP2) [D:1.00; C:1.00]	2: (AQ2) [1]
4.0000: (A*) [D:1.00; C:1.00]	3: (AQ3) [1]
5.0000: (AP3) [D:1.00; C:1.00]	4: (A*) [1]
6.0000: (A*) [D:1.00; C:1.00]	
7.0000: (A1+) [D:1.00; C:1.00]	
LPG-ADAPT **plan**	ERRT-PLAN **plan**

Figure 9: Plans generated by LPG-ADAPT and ERRT-PLAN on the problem.

In fact ap1+ needs seven steps for achieving p3, but those steps are hidden by the deletion of g* in all actions, as well as deleting the previously achieved proposition. This was the main difference between the behavior of total-order nonlinear planners, as PRODIGY, and partial-order planners, as UCPOP. The latter tried to eagerly achieve open preconditions with actions from the current partial plan, failing in this case. We have presented the simplest case where the previous plan was composed of only one action and one goal, g1. But, the same phenomena happens when the previous solutions have another thousand actions in the plan achieving a set of goals of any size, and we made the small change in the goals of changing g1 for g+. Even if this domain seems far fetched, consider now that g* is a renewable resource as fuel, and then it matches the behavior of many real world domains. Again, the configurations of ERRT-PLAN are able

to find the right solution, given that for it the previous plan is not the starting point of a local search, but a guidance that can be used or not as needed.

Combining Reuse of Past Plans with Planning

We have also performed experiments in the Satellite and Rovers IPC domains,[10] In Satellite, we used the given 20 IPC problems, and generated three variations of each problem by adding a random goal (so, the total number of problems per domain was 60). Then, we solved each original problem with FF for a seed plan and used its solution for solving each newly generated problem with FF and each ERRT-PLAN configuration.

In Figure 10 we show the planning time of FF and ERRT-PLAN in the last problem of the IPC of the Satellite domain, pfile20, given that the rest of problems were solved in less than 1 second. We run each ERRT-PLAN configuration five times and show the median. The configuration with $r_a = 1.0$ could not solve the problems, while the rest could solve most of the runs (114 runs out of 123). As we can see, the configurations with no action reuse ($r_a = 0.0$) have worse performance, while the rest are two orders of magnitude better than FF.

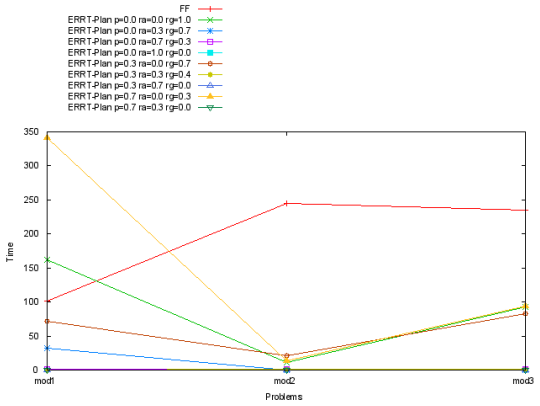


Figure 10: Median of the planning time of five runs for three modifications of pfile20 of the Satellite domain.

In Figure 11 we show the results on the Rovers domain. We used in this case four problems of the IPC (p30 to p33) as representatives of medium complexity problems (solving time greater than a few seconds and less than 600 seconds). The setting is the same as the previous experiments. Again, we see the same behavior as in the Satellite domain: configurations with no action reuse have bad performance, while the rest can obtain up to two orders of magnitude improvement over FF.

In Figure 12 we show the results on the Zenotravel domain of the most difficult problems of the competition, pfile16

[10] International Planning Competition.

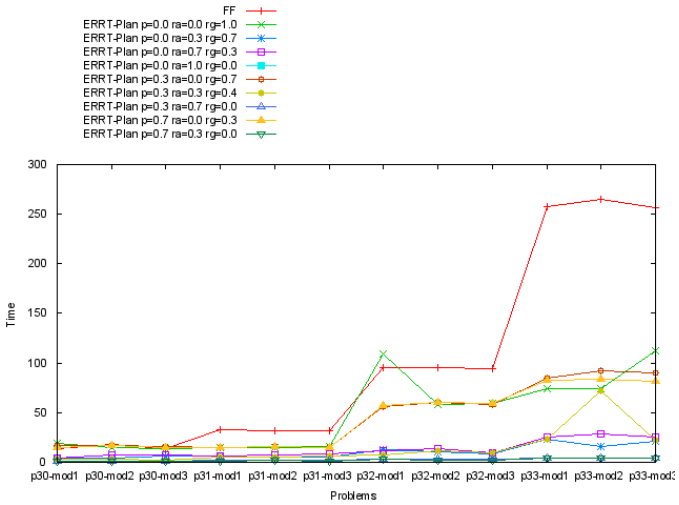


Figure 11: Median of the planning time of five runs for three modifications of problems p30 to p33 of the Rovers domain.

to pfile20. And we see the same behavior as in the previous domains: two orders of magnitude improvement on planning time and worse results by $r_a = 0.0$ configurations.

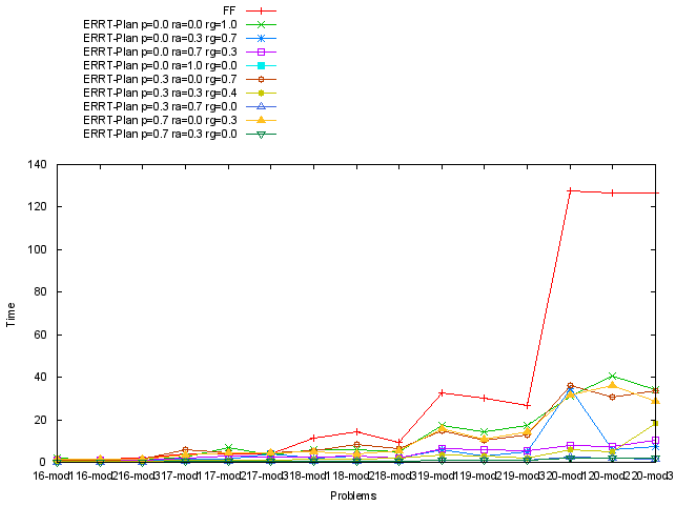


Figure 12: Median of the planning time of five runs for three modifications of problems pfile16 to pfile20 of the Zenotravel domain.

In summary, we can see how using both reuse approaches at the same time improves over using only one of them ($r_a = 0.0$ or $r_g = 0.0$). In general, using action-reuse is a better strategy than using goal-reuse. The reason is that, in the extreme, goal-reuse ends up being almost the same as EHC given that when the goal-reuse pointer reaches the end of the previous plan, there are no more goals to point to, and the goals of the new problem are always selected. But, action-reuse by itself does not have a good behavior either, because once actions of the previous plan have been reused, we still might need to do some extra work to achieve the new goals.

So, the combination of both goal- and action-reuse makes the approach robust and compensate EHC drawbacks. Also, we can see that almost all configurations use much less time than pure EHC: two orders of magnitude less in many cases.

Conclusions and Future Work

We have presented a new approach to planning that stochastically uses either a greedy planning heuristic or prior knowledge in the form of actions or goals to achieve. Given that nor current planning heuristics and search techniques, nor previous knowledge (plans) are proved to be always correct, we use them as biases, and use probability values that drive the search direction. Results show that our ERRT-PLAN approaches are able to successfully address difficult open questions in planning such as how to efficiently use knowledge of a previous plan to solve new similar problems. This work opens further avenues of research, including the use of any other search technique, instead of programming it on top of EHC as best-first techniques. In addition, any multiple biases can be stochastically included in ERRT-PLAN, such as different heuristics or specific human guidance.

Acknowledgements

This work has been partially supported by MICINN projects TIN2008-06701-C03-03 and TIN2011-27652-C03-02. This research was also partially sponsored by the Office of Naval Research under grant number N00014-09-1-1031. The views and conclusions of this work are those of the authors only.

References

Alcázar, V.; Veloso, M.; and Borrajo, D. 2011. Adapting an RRT for automated planning. In Borrajo, D.; López, C. L.; and Likhachev, M., eds., *Proceedings of the Fourth International Symposium on Combinatorial Search (SoCS-2011)*, 2–9. Cardona (Spain): AAAI Press.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116:123–191.

Bruce, J., and Veloso, M. 2002. Real-time randomized path planning for robot navigation. In *Proceedings of IROS-2002*.

Burfoot, D.; Pineau, J.; and Dudek, G. 2006. RRT-Plan: A Randomized Algorithm for STRIPS Planning. In *Proceedings of ICAPS'06*, 362–365. Ambleside (UK): AAAI Press.

De la Rosa, T.; García-Olaya, A.; and Borrajo, D. 2007. Using cases utility for heuristic planning improvement. In Weber, R., and Richter, M., eds., *Case-Based Reasoning Research and Development: Proceedings of the 7th International Conference on Case-Based Reasoning*, volume 4626 of *Lecture Notes on Artificial Intelligence*, 137–148. Belfast, Northern Ireland, UK: Springer Verlag.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. In *Proceedings of ICAPS'06*, 212–221.

Helmert, M., and Roger, G. 2008. How good is almost perfect. In *Proceedings of AAAI'08*, 944–949.

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

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research* 20:291–341.

Kambhampati, S., and Hendler, J. A. 1992. A validation based theory of plan modification and reuse. *Artificial Intelligence* 55(2-3):193–258.

LaValle, S. M., and Kuffner, J. J. J. 2001. Randomized kinodynamic planning. In *Proceedings of the International Journal of Robotics Research*, volume 20, 378–400.

Nau, D.; Au, T.-C.; Ilghami, O.; Kuter, U.; Mur, J. W.; and Wu, D. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research* 20:379–404.

Nebel, B., and Koehler, J. 1995. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence* 76:427–454.

Serina, I. 2010. Kernel functions for case-based planning. *Artificial Intelligence* 174:1369–1406.

Veloso, M. M., and Blythe, J. 1994. Linkability: Examining causal link commitments in partial-order planning. In *Proceedings of the Second International Conference on AI Planning Systems*, 170–175. Chicago, IL: AAAI Press, CA.

Veloso, M. M., and Carbonell, J. G. 1993. Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning* 10:249–278.

Preferring Properly: Increasing Coverage while Maintaining Quality in Anytime Temporal Planning

Patrick Eyerich
Albert-Ludwigs-Universität Freiburg
Institut für Informatik
Georges-Köhler-Allee 52
79110 Freiburg, Germany
eyerich@informatik.uni-freiburg.de

Abstract

Temporal Fast Downward (TFD) is a successful temporal planning system that is capable of dealing with numerical values. Rather than decoupling action selection from scheduling, it searches directly in the space of time-stamped states, an approach that has shown to produce plans of high quality at the price of coverage. To increase coverage, TFD incorporates deferred evaluation and preferred operators, two search techniques that usually decrease the number of heuristic calculations by a large amount. However, the current definition of preferred operators offers only limited guidance in problems where heuristic estimates are weak or where subgoals require the execution of mutex operators. In this paper, we present novel methods of how to refine this definition and show how to combine the diverse strengths of different sets of preferred operators using a restarting procedure incorporated into a multi-queue best-first search. These techniques improve TFD's coverage drastically and preserve the average solution quality, leading to a system that solves more problems than each of the competitors of the temporal satisficing track of IPC 2011 and clearly outperforms all of them in terms of IPC score.

Introduction

Temporal Planning is an important generalization of classical planning that allows to model many applications more realistically by taking into account not only causal dependencies between actions but also their temporal interactions. It is a growing research area and there are many interesting approaches that tackle its challenges. LPG (Gerevini, Saetti, and Serina 2003) is based on stochastic local search in the space of *action graphs*. Crikey3 (Coles et al. 2008) employs heuristic forward search interleaving planning and scheduling via Simple Temporal Networks. CPT4 (Vidal 2011a) is a planning system based on partial order causal links that is optimal for the conservative semantics of Smith and Weld (Smith and Weld 1999). A common approach to temporal planning, as for example taken by SGPlan (Hsu and Wah 2008), YAHSP2 (Vidal 2011b) and $\text{DAE}_{\text{YAHSP}}$ (Dréo et al. 2011), is to consider only logical dependencies between actions first while temporal dependencies are taken into account just afterwards to find an appropriate scheduling for the chosen actions. While having the potential of being very fast due to the possibility of utilizing successful techniques from the much more investigated field of classical planning, such approaches are doomed to fail in temporally expressive domains (Cushing et al. 2007), and suffer from severe drawbacks in temporally simple problems, as choosing the wrong actions might render the final solutions to be purely sequential and therefore of very low quality.

Another approach as for example taken by Sapa (Do and Kambhampati 2003b), LMTD (Hu, Cai, and Yin 2011), or Temporal Fast Downward (TFD) (Eyerich, Mattmüller, and Röger 2009), is to perform forward search in the space of time-stamped states, where at each search state either a new action can be started or time can be advanced to the end point of an already running action, thereby combining action selection and scheduling. Also, POPF2 (Coles et al. 2011; 2010) performs a forward search, using a partial order rather than a total order like Sapa and TFD do. While these approaches are usually very good in terms of quality, their coverage on current benchmarks is typically relatively low.

As a first step to increase its coverage, TFD incorporates preferred operators and deferred evaluation (Richter and Helmert 2009). The general idea of preferred operators is to favor operators that are part of a solution for the heuristic abstraction of the problem. Deferred evaluation postpones heuristic computations from the point where a search node is generated to the point where it is expanded, rating nodes with the estimate of their predecessor during search. Thereby, the number of heuristic calculations is decreased by a large amount at the price of informativeness. Even more than in classical planning, in temporal planning the heuristic computation is the bottleneck of search, and indeed it turns out that the use of deferred evaluation and preferred operators increases the performance of TFD enormously. Unfortunately, this improvement does not occur uniformly over all planning domains. Instead, there are problems where using preferred operators and deferred evaluation worsens results.

The first contribution of this paper consists in novel methods that strengthen the selection criteria for preferred operators. Different methods have strengths on different domains and some of them clearly increase TFD's coverage on their own. On their downside, all new methods have in common that they produce solutions of lower quality than the original definition. This leads to the second contribution, a restarting procedure embedded in a multi-queue best first search that, besides further increasing coverage, regains the lost quality. Our resulting system is able to overcome the disadvantage of

searching in the space of time stamped states, i.e., low coverage, while maintaining its major advantage, high solution quality.

The remainder of this paper is structured as follows: In the next section we describe TFD with an emphasis on its heuristic. The subsequent section presents our novel selection criteria for preferred operators and a multi-queue search algorithm featuring restarts. Afterwards, we present detailed experiments before we conclude. Related work is referred to throughout the paper whenever it fits.

Temporal Fast Downward

Temporal Fast Downward (TFD) (Eyerich, Mattmüller, and Röger 2009) is a domain-independent progression search planner 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. TFD solves a planning task in three phases: As a first step, the Boolean PDDL encoding is *translated* into a finite-domain representation similar to SAS+ (Bäckström and Nebel 1996; Helmert 2009). Afterwards, in a *knowledge compilation* step, several internal data structures are generated. The scope of this paper is the third part, a best-first progression *search*. For the sake of simplicity, we only deal with non-numerical fluents. Note, however, that all our results can be generalized to the numeric case straightforwardly.

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 durative action $a = \langle C, E, \delta \rangle$ consists of a triple $C = \langle C_\vdash, C_\leftrightarrow, C_\dashv \rangle$ of partial states (called its *start*, *persistent*, and *end condition*, respectively), a tuple $E = \langle E_\vdash, E_\dashv \rangle$ of *start* and *end effects*, and a duration variable $\delta \in \mathcal{V}$. $E_\vdash$ and $E_\dashv$ are finite sets of *conditional effects* $\langle c, e \rangle$. Their *effect condition* $c = \langle c_\vdash, c_\leftrightarrow, c_\dashv \rangle$ is defined analogously to C. A *simple effect* e is of form $v{=}w$.

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 scheduled effect $\langle \Delta t, c_\leftrightarrow, c_\dashv, e \rangle$ consists of the remaining time $\Delta t \geq 0$ (until the instant when the effect triggers), persistent and end effect conditions $c_\leftrightarrow$ and $c_\dashv$ over $\mathcal{V}$, and a simple effect e. The conditions in $C_\leftrightarrow$ and $C_\dashv$ are annotated with time increments $\Delta t \geq 0$ and have to hold until instant $t + \Delta t$ (exclusively) for persistent conditions and at instant $t + \Delta t$ for 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.

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:

$$h^{cea}(x|x') \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x = x' \\ \min\limits_{o: x'', z \rightarrow x} \big(c(o, s'') + & \\ \quad h^{cea}(x''|x') + & \\ \quad \sum\limits_{x_i \in z} h^{cea}(x_i|x_i'')\big) & \text{else} \end{cases}$$

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 2003a; 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 have 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

$$\mathcal{P}(x|x') \stackrel{\text{def}}{=} \begin{cases} \{\} & \text{if } x = x' \text{ or } h^{cea}(x|x') = \infty \\ \{o\} & \text{if } \exists\, o : x', w \to x : \\ & \quad h^{cea}(x|x') = c(o, s') \\ \bigcup\limits_{x_i \in w} \mathcal{P}(x_i|x_i') & \text{if } \exists\, o : x', w \to x : \\ & \quad h^{cea}(x|x') = \Big(c(o, s') + \\ & \quad \sum\limits_{x_i \in w} h^{cea}(x_i|x_i')\Big) \\ \mathcal{P}(x''|x') & \text{if } \exists\, x'' : h^{cea}(x''|x') \\ & \quad + h^{cea}(x|x'') = h^{cea}(x|x') \end{cases}$$

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.

The first case is trivial. The second case defines an operator o that transforms x' to x with cost equal to $h^{cea}(x|x')$ (which means that all its preconditions have to be satisfied) as preferred. The third case is similar to the second one except that some of the operator's preconditions are not satisfied. In that case, preferred operators are recursively defined over these 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). As can be seen in the experiments section, 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. Things are analogously for expanding s_2. In such a situation a potentially very large set of states has to be visited before the search actually progresses in the right direction.

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 useful 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 restrict ourselves to the natural ordering that is defined by the order in which variables occur in the problem description and defer the interesting question of how to combine our technique with goal ordering detection techniques to future work.

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 (lines 1 and 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 initially boosted 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.

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
```

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.

Experiments

In our first experiment[1] we show the influence that deferred evaluation and preferred operators have on the search performance of TFD.

Results showing IPC score[2] and coverage (in parentheses) on IPC 2011 benchmarks are presented in Table 1. Without preferred operators, switching from eager ('e') to deferred evaluation ('d') speeds up the search by saving a lot of heuristic computations but reduces guidance, altogether

[1] All our experiments were run on AMD Opteron 2.3 GHZ Dual Core processors with a memory limit of 2 GB and a timeout of 30 minutes.

[2] If Q^* is the makespan of a reference solution, a planner producing a solution of makespan Q receives Q^*/Q points of IPC score. For all our experiments the best known plans (including ours) are used as reference.

IPC 2011	e	d	$\mathcal{P}$e	$\mathcal{P}$d
CREWPLANNING	0.0 (0)	0.0 (0)	**19.9** (**20**)	**19.9** (**20**)
ELEVATORS	0.0 (0)	0.0 (0)	0.0 (0)	**1.0** (**1**)
FLOORTILE	4.0 (**5**)	4.1 (**5**)	**4.9** (**5**)	**4.9** (**5**)
MATCHCELLAR	1.0 (1)	1.0 (1)	**15.6** (**20**)	**15.6** (**20**)
OPENSTACKS	**17.8** (**20**)	16.7 (**20**)	**17.8** (**20**)	17.7 (**20**)
PARCPRINTER	0.0 (0)	0.0 (0)	0.0 (0)	0.0 (0)
PARKING	**13.7** (**17**)	10.2 (14)	7.1 (9)	12.2 (16)
PEGSOL	17.9 (**18**)	**18.0** (**18**)	17.9 (**18**)	17.9 (**18**)
SOKOBAN	**2.9** (**3**)	**2.9** (**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	12.0 (14)	10.7 (13)	12.5 (17)	**13.3** (**20**)
Overall	69.2 (78)	63.7 (74)	98.6 (112)	**105.3** (**123**)

Table 1: Results on IPC 2011 benchmarks measuring the influence of deferred evaluation and preferred operators in regular TFD, showing IPC score and coverage (in parentheses). Used abbreviations are '$\mathcal{P}$' for preferred operators, 'd' for deferred evaluation, and 'e' for eager evaluation.

leading to a slightly worse performance. This drawback, however, can be overcome by incorporating preferred operators ('$\mathcal{P}$d'). With preferred operators bringing back some of the lost guidance, the advantages of the reduced computational effort of deferred evaluation can be fully exploited, leading to both a much higher coverage and IPC score. Note that using preferred operators increases performance also in the context of eager evaluation ('$\mathcal{P}$e'), but to a lesser degree, so that combining preferred operators and deferred evaluation clearly is the best option.

For our second experiment we have implemented the methods presented in Section to narrow the set of preferred operators. Results for $\mathcal{C}^n$, $\mathcal{E}^n$, and $\mathcal{O}^n$ with $1 \leq n \leq 3$ are shown in Table 2.

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 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.

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 2: Performance of new selection strategies on IPC 2011 benchmarks. Gray rows show IPC scores, white rows coverage.

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 3 shows results of an implementation of Algorithm 1 combining narrowing strategies with a queue containing the original preferred operators ($\mathcal{P}$). The method that profits the most from this combination is $\mathcal{O}^1$, with the most important factor being the gain in CREWPLANNING. Besides, the other versions profit also, especially in terms of quality. Both $\mathcal{PC}$ and $\mathcal{PO}$ achieve higher IPC scores than $\mathcal{P}$ alone. Table 4 shows that the power of combining selections strategies can be exploited even further, with $\mathcal{PO}^1\mathcal{C}^1\mathcal{E}^1$ achieving both the highest coverage and IPC score.

IPC 2011	$\mathcal{PC}^1$	$\mathcal{PC}^2$	$\mathcal{PC}^3$	$\mathcal{PE}^1$	$\mathcal{PE}^2$	$\mathcal{PE}^3$	$\mathcal{PO}^1$	$\mathcal{PO}^2$	$\mathcal{PO}^3$
CREWPLANNING	**19.9**	**19.9**	**19.9**	**19.9**	**19.9**	**19.9**	**19.9**	**19.9**	**19.9**
	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
ELEVATORS	**15.1**	7.7	6.5	0.0	0.0	0.0	13.4	4.3	0.9
	18	9	7	0	0	0	**18**	6	1
FLOORTILE	4.6	**4.7**	4.5	4.4	4.5	4.3	4.5	4.5	4.4
	5	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
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	17.9	18.0	18.3	18.5	**18.7**	18.6	18.0	17.7	18.1
	20	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**
PARCPRINTER	1.0	1.0	1.8	0.0	0.0	0.0	**9.3**	0.9	0.0
	1	1	2	0	0	0	**10**	1	0
PARKING	13.9	**15.7**	13.4	10.9	11.5	11.1	11.9	12.9	12.3
	18	**20**	17	15	15	15	16	17	16
PEGSOL	17.9	17.9	18.5	17.9	18.4	18.5	**18.7**	18.5	18.6
	18	18	**19**	18	**19**	**19**	**19**	**19**	**19**
SOKOBAN	2.9	2.9	2.9	**3.0**	2.9	2.9	**3.0**	2.9	2.9
	3	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**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	13.9	13.2	13.2	13.0	13.4	12.9	**14.2**	13.7	12.6
	20	19	19	18	19	18	**20**	19	18
Overall	122.6	116.6	114.5	103.1	104.8	103.8	**128.4**	110.8	105.3
	143	135	132	119	121	120	**151**	130	122

Table 3: Combining narrowing strategies with the original selection strategy ($\mathcal{P}$) via restarting as described in Algorithm 1. Gray rows show IPC scores, white rows coverage.

To see how these improvements affect the performance of TFD relatively to other temporal planning systems, we compared both the original TFD and TFD enriched with our new techniques 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.[3] Table 5 presents IPC scores (gray rows) and coverage (white rows). It can be seen that $\mathcal{PO}^1\mathcal{C}^1\mathcal{E}^1$ 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.

[3]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	$\mathcal{PC}^1\mathcal{O}^1$	$\mathcal{PC}^1\mathcal{E}^1$	$\mathcal{PO}^1\mathcal{E}^1$	$\mathcal{PO}^1\mathcal{C}^1\mathcal{E}^1$
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 4: IPC scores and coverage (in parentheses) of combining more than one narrowing strategy via restarting as described in Algorithm 1.

In another experiment, presented in Table 6, 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 only makespan but a more complex metric needs to be optimized, a feature TFD cannot deal with yet). Results are presented in Table 7. Note that only for the benchmark set of 2008 reference plans are used. In this experiment the title of this paper is reflected very well: Coverage is increased drastically while the average plan quality is 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, leading to a system that solves more problems than each of the competitors of the temporal satisficing track of IPC 2011 and clearly outperforms all of them in terms of IPC score. Furthermore, we have shown that these excellent behavior also occurs on the benchmark suites of 2006 and 2008. Future work includes incorporating goal ordering techniques to find more sophisticated orderings for $\mathcal{O}$ as well as determining additional selection strategies for

preferred operators that might increase the coverage of TFD even further. While this work is motivated from the large gap between coverage and quality when searching in the space of time-stamped states, it can also be applied to classical planning and doing so is a major part of our future work.

IPC 2011	CPT4	LMTD	YAHSP2	YAHSP2-mt	POPF2	DAE-YAHSP	TFD	$\mathcal{PO}^1\mathcal{C}^1\mathcal{E}^1$
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 5: Gray rows show IPC scores, white rows coverage of participants of IPC 2011 that solved at least one instance. The two rightmost columns show results of TFD using the original selection strategy for preferred operators and using our new techniques ($\mathcal{PO}^1\mathcal{C}^1\mathcal{E}^1$).

IPC 2011	CPT4	LMTD	Y2	Y2-mt	POPF2	DAE-Y	TFD
CREWPLANNING	7**1.00**	–	20**1.29**	20**1.29**	200.99	200.99	20**1.00**
ELEVATORS	–	90.94	18**2.08**	18**1.98**	3**1.01**	150.93	10.59
FLOORTILE	5**1.67**	20.99	4**2.38**	5**2.22**	–	4**2.36**	50.96
MATCHCELLAR	–15**1.06**	–	–	20**1.25**	–	20**1.24**	
OPENSTACKS	–	–	20**1.47**	19**1.46**	20**1.23**	200.92	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	180.99	190.98	19**1.16**	19**1.11**	18**1.00**	190.98	18**1.00**
SOKOBAN	–	–	3**1.02**	3**1.03**	2**1.17**	3**1.10**	3**1.00**
STORAGE	–	–	–	–	–	–	–
TMS	–	–	–	–	–	–	–
TURN AND OPEN	–13**1.38**		–	–	90.61	–	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 6: Pairwise plan quality comparisons to TFD$^+$ featuring our new techniques, namely separate queues $\mathcal{O}^1$, $\mathcal{C}^1$, and $\mathcal{E}^1$, respectively, and restarting like described in Algorithm 1. 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 7: 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.

References

Bäckström, C., and Nebel, B. 1996. Complexity Results for SAS^+ Planning. *Computational Intelligence* 11:625–655.

Coles, A.; Fox, M.; Long, D.; and Smith, A. 2008. Planning with Problems Requiring Temporal Coordination. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, 892–897.

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.

Coles, A.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-Chaining Partial-Order 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*, 42–49.

Coles, A.; Coles, A.; Clark, A.; and Gilmore, S. 2011. Cost-Sensitive Concurrent Planning under Duration Uncertainty for Service Level Agreements. In Bacchus, F.; Domshlak, C.; Edelkamp, S.; and Helmert, M., eds., *Proceedings of the Twenty First International Conference on Automated Planning and Scheduling*, 34–41.

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. S. 2007. When is Temporal Planning Really Temporal? In Veloso, M. M., ed., *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 1852–1859.

Do, M. B., and Kambhampati, S. 2003a. 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.

Do, M. B., and Kambhampati, S. 2003b. Sapa: A Multi-objective Metric Temporal Planner. *Journal of Artificial Intelligence Research* 20:155–194.

Dréo, J.; Savéant, P.; Schoenauer, M.; and Vidal, V. 2011. Divide-and-Evolve: The Marriage of Descartes and Darwin. In Angel García-Olaya, S. J., and López, C. L., eds., *Seventh International Planning Competition*, 29–30.

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.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning Through Stochastic Local Search and Temporal Action Graphs in LPG. *Journal of Artificial Intelligence Research* 20:239–290.

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.

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence* 173:503–535.

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

Hsu, C., and Wah, B. W. 2008. The SGPlan Planning System in IPC-6.

Hu, Y.; Cai, D.; and Yin, M. 2011. The LMTD Planner: On the Discovery and Utility of Precedence Constraints in Temporal Planning. In Angel García-Olaya, S. J., and López, C. L., eds., *Seventh International Planning Competition*, 128–131.

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.

Vidal, V. 2011a. CPT4: An Optimal Temporal Planner Lost in a Planning Competition without Optimal Temporal Track. In Angel García-Olaya, S. J., and López, C. L., eds., *Seventh International Planning Competition*, 25–28.

Vidal, V. 2011b. YAHSP2: Keep it Simple, Stupid. In Angel García-Olaya, S. J., and López, C. L., eds., *Seventh International Planning Competition*, 83–90.

Domain-Independent Relaxation Heuristics for Probabilistic Planning with Dead-ends

Florent Teichteil-Königsbuch and **Vincent Vidal** and **Guillaume Infantes**
name.surname@onera.fr
ONERA — The French Aerospace Lab
F-31055, Toulouse, France

Abstract

Recent domain-determinization techniques have been very successful in many probabilistic planning problems. We claim that traditional heuristic MDP algorithms have been unsuccessful due mostly to the lack of efficient heuristics in structured domains. Previous attempts like mGPT used classical planning heuristics to an all-outcome determinization of MDPs without discount factor ; yet, discounted optimization is required to solve problems with potential dead-ends. We propose a general extension of classical planning heuristics to goal-oriented discounted MDPs, in order to overcome this flaw. We apply our theoretical analysis to the well-known classical planning heuristics h_{max} and h_{add}, and prove that the extended h_{max} is admissible. We plugged our extended heuristics to popular graph-based (Improved-LAO*, LRTDP, LDFS) and ADD-based (sLAO*, sRTDP) MDP algorithms: experimental evaluations highlight competitive results compared with the winners of past competitions (FF-REPLAN, FPG, RFF), and show that our discounted heuristics solve more problems than non-discounted ones, with better criteria values. As for classical planning, the extended h_{add} outperforms the extended h_{max} on most problems.

Introduction

Significant progress in solving large goal-oriented probabilistic planning problems has been achieved recently, partly due to tough challenges around the probabilistic part of International Planning Competitions (Younes et al. 2005). Looking back at successful planners, most of them rely on a deterministic planner to solve the probabilistic planning problem: FPG-FF (Buffet and Aberdeen 2007), FF-REPLAN (Yoon, Fern, and Givan 2007), RFF (Teichteil-Königsbuch, Kuter, and Infantes 2010), FF-H (Yoon et al. 2010), GOTH (Kolobov, Mausam, and Weld 2010). These planners "determinize" the original domain, generally by replacing probabilistic effects by the most probable one for each action ("most probable outcome" determinization), or by considering as many deterministic actions as probabilistic effects ("all-outcomes" determinization). Other successful but non determinization-based approaches are FPG (Buffet and Aberdeen 2009) and FODD-PLANNER (Joshi, Kersting, and Khardon 2010). It is noteworthy to mention that FPG was later improved to FPG-FF using a deterministic planner to guide simulated trajectories to the goal.

Yet, to our knowledge, there does not really exist any domain-independent, optimal and fully probabilistic algorithm whose performances are competitive with determinization-based approaches. In particular, traditional Markov Decision Process (MDP) heuristic algorithms like (s)LAO* (Hansen and Zilberstein 2001; Feng and Hansen 2002), (s,L)RTDP (Bonet and Geffner 2003; Feng, Hansen, and Zilberstein 2003), or LDFS (Bonet and Geffner 2006), have not enjoyed competitive results on the competition domains. However, one can wonder whether the efficiency of determinization-based planners is due to solving many simpler deterministic problems, or rather to very efficient heuristics implemented in the underlying deterministic planner.

The mGPT planner by (Bonet and Geffner 2005) partially answered this question; by applying state-of-the-art classical planning heuristics to an all-outcome determinization, it achieved good performances on domains modeled as Stochastic Shortest Path Problems (SSPs). As formalized later, such problems assume that there exists an optimal policy that reaches a goal state with probability 1. If not, it means that any optimal policy may reach some states with a positive probability, from where no goal states are then reachable (called *dead-ends*). In the 2004 probabilistic competition, mGPT could not solve domains with dead-ends like exploding blocksworld.

In this paper, we propose to handle dead-ends by means of a *discounted* criterion, thus solving *discounted SSPs*. Our work is different from the inverse transformation done in (Bertsekas and Tsitsiklis 1996) where it is shown that any γ-discounted MDP *without termination state* can be stated as an equivalent SSP (without discount factor): in their work, the goal state of the SSP only models the anytime termination of the process with probability $1 - \gamma$, and is defined as an absorbing state that is reachable from any state. The main drawbacks of this approach are: (1) the goal states of the original problem cannot be related to any state of the equivalent SSP, so that an algorithm for the equivalent SSP cannot be guided towards the goal states of the problem; (2) dead-ends of the original problem lead themselves to the goal state of the equivalent SSP, thus trying to reach the goal state leads to a policy that is attracted by dead-ends. In order to avoid

these flaws, in our work, we rather reason about the true goal states of the original problem, which is seen as a SSP whose costs are discounted.

Therefore, we propose (1) a generic extension of any classical planning heuristic to classical SSPs *without dead-end* and (2) a further extension to discounted SSPs (able to deal with dead-ends) that directly target the original problem's goal states. We apply these approaches to the well-known h_{max} and h_{add} heuristics by (Bonet and Geffner 2001), but could have extended other classical planning heuristics like h_{FF} by (Hoffmann 2001) in a similar way. We use our heuristics in state-of-the-art MDP heuristic search algorithms (`Improved-LAO*`, `LRTDP`, `LDFS`, `sLAO*`, `sRTDP`) with discounted settings, which always converge in presence or not of reachable dead-ends. We experimentally show that these algorithms with our discounted heuristics provide better goal reaching probability and average length to the goal than the winners of previous competitions or the same algorithms without discount factor.

Goal-oriented Markov Decision Process

The following definition is slightly adapted from (Wu, Kalyanam, and Givan 2008).

Definition 1. A goal-oriented MDP is a tuple $\langle \mathcal{S}, \mathcal{I}, \mathcal{G}, \mathcal{A}, app, T, C \rangle$ with:

- $\mathcal{S}$ a set of states;
- $\mathcal{I} \subseteq \mathcal{S}$ a set of possible initial states;
- $\mathcal{G} \subseteq \mathcal{S}$ a set of goal states;
- $\mathcal{A}$ a set of actions;
- $app : \mathcal{S} \to 2^{\mathcal{A}}$ an applicability function: $app(s)$ is the set of actions applicable in s;
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0; 1]$ a transition function such that $T(s, a, s') = Pr(s' \mid a, s)$ and $T(g, a, s') = 0$ for all $g \in \mathcal{G}$ and $s' \notin \mathcal{G}$ (goal states are absorbing);
- $C : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_+$ a cost function such that, for all $a \in \mathcal{A}$, $C(s, a) = 0$ if $s \in \mathcal{G}$, and $C(s, a) > 0$ if $s \notin \mathcal{G}$.

A solution of a goal-oriented MDP is a partial policy $\pi_{\mathcal{X}} : \mathcal{X} \subseteq \mathcal{S} \to \mathcal{A}$ mapping a subset of states $\mathcal{X}$ to actions minimizing some criterion based on the costs. $\mathcal{X}$ contains all initial states, at least one goal state. Moreover, $\pi_{\mathcal{X}}$ is closed: states reachable by applying $\pi_{\mathcal{X}}$ on any state in $\mathcal{X}$ are in $\mathcal{X}$.

STRIPS representation of MDPs

ADL representation of MDPs as formalized in the PPDDL language by (Younes et al. 2005), has facilitated benchmark sharing and planners comparison. Moreover, this representation can be transformed into a simpler STRIPS form (Gazen and Knoblock 1997), that enabled to derive efficient domain-independent heuristics from the model, which has been mainly exploited by determinization-based approaches through the underlying deterministic planner. Yet, as partially highlighted by (Bonet and Geffner 2005), we claim that efficient domain-independent heuristics can be directly derived from the original probabilistic domain without determinization, and plugged to MDP heuristic algorithms.

In probabilistic STRIPS, states can be seen as collections of atoms from the set Ω of all atoms. Actions are probabilistic operators of the form $o = \langle prec, cost, [p_1 : (add_1, del_1), \cdots, p_m : (add_m, del_m)] \rangle$ where $\sum_{i=1}^{m} p_i = 1$ (more details in (Younes et al. 2005; Bonet and Geffner 2005)) and:

- $prec$ is a collection of atoms such that the action is applicable in a state s iff $prec \subseteq s$;
- $cost$ is the cost of the action; for each $i \in [1; m]$, p_i is the probability of the i^{th} effect which is represented by the set add_i of atoms becoming true and the set del_i of atoms becoming false; atoms that are neither in add_i nor in del_i keep their values.

Note that the remaining of this paper only assumes a probabilistic STRIPS representation of goal-oriented MDPs, so our contribution is valid for any formalism that boils down to such representation.

We illustrate this concept with the well-known `blocksworld` problem, which consists in building an ordered stack of n blocks $(b_1, \cdots, b_n)$ by a robot with a single hand, where each block can initially belong to different stacks. There are $n(n + 3) + 1$ atoms: $(emptyhand)$; $(holding\ b)$, $(ontable\ b)$ and $(clear\ b)$ for each block b; $(on\ b_1\ b_2)$ for any two blocks b_1 and b_2. As an example of action, $(pickup\ b_1\ b_2)$ is defined by:

$$\langle \{(emptyhand), (clear\ b_1), (on\ b_1\ b_2)\}, \\ [0.75 : (\{(holding\ b_1), (clear\ b_2)\}, \\ \{(emptyhand), (on\ b_1\ b_2)\}) \\ 0.25 : (\{(clear\ b_2), (ontable\ b_1)\}, \\ \{(on\ b_1\ b_2)\})] \rangle$$

Dead-end states

The existence of a solution depends on structural properties of the goal-oriented MDP, precisely on the presence of reachable dead-ends, as defined below.

Definition 2. A *dead-end* is a state from which the probability to reach the goal with any policy is equal to zero.

A non-goal absorbing state is a dead-end state. Now, consider the following criterion, known as the *total criterion*:

$$\forall s \in \mathcal{S}, \pi^*(s) = \text{argmin}_{\pi \in \mathcal{A}^S} E\left[\sum_{t=0}^{+\infty} C_t \mid s_0 = s, \pi\right] \quad (1)$$

where C_t is the cost received at time t starting from s and executing policy π. If π can reach some dead-end state s_d with a positive probability, as no goal state is reachable from s_d by definition, and because costs received in any non-goal state are strictly positive, the sum of future costs is $+\infty$. Thus, *eq. 1 has a solution iff there exists at least one policy that, with probability 1, does not reach any dead-end state*.

An example of goal-oriented MDP with dead-ends is the `exploding blocksworld` domain, which extends the `blocksworld` domain to blocks that can detonate and then destroy other blocks or the table itself. The domain contains $2n + 1$ additional atoms: $(nodetonated\ b)$ and $(nodestroyed\ b)$ for each block b; $(nodestroyed - table)$. To build a goal stack, two kinds of actions are required: putting a block on the table, which can detonate the block and destroy the table with probability 3/5; putting a block

on another block, which can detonate the handled block and destroy the target block with probability 1/10. Destroying the table or a block belonging to a goal stack prevents from reaching the goal, and thus all states reachable from these situations are dead-ends. As all policies need to apply such actions to reach the goal, these dead-ends are reachable with a positive probability by executing any policy from the initial state.

Whether the total criterion has a solution or not is unrelated to the general problem of finding a policy that reaches a goal state with a positive probability (possibly not 1), whatever the minimization criterion considered. It only means that a planner based on the total criterion, as `mGPT` is, will not find a working solution for any domain with reachable dead-ends like `exploding blocksworld`. But other planners that are not based on this criterion, or that do not optimize any criterion like `FF-REPLAN`, will possibly find a solution for this domain, that reaches a goal state with a probability lower than 1. And this is the expected result.

Unfortunately, deciding if a given domain contains dead-ends reachable by the optimal policy boils down to optimizing the total criterion itself. Thus, most algorithms based on this criterion, like `mGPT`, simply cross the fingers: they try to solve the problem and hope for the best. Yet, the total criterion is very popular in heuristic MDP planning because it allows to design efficient domain-independent admissible heuristics, as explained in the next section.

Solving *undiscounted* goal-oriented MDPs

This class of MDPs, known as Stochastic Shortest Path Problems (SSPs) extended to positive costs, has been extensively studied[1]. If there exists an optimal policy reaching a goal state with probability 1, the total criterion of eq. 1 is well-defined, and heuristic algorithms optimize only relevant states when starting from known initial states. A forward-chaining procedure iteratively expands the searched state space until the value of explored states has converged. The convergence condition actually depends on the way states are explored, i.e. on the heuristic and on the algorithm exploiting it. Yet, all these algorithms update the value of any explored state s by using the same Bellman backup equation (see (Bonet and Geffner 2003; 2006; Hansen and Zilberstein 2001)):

$$V(s) \leftarrow \min_{a \in app(s)} \left\{ C(s,a) + \sum_{s' \in \mathcal{S}} T(s,a,s') \widetilde{V}(s') \right\} \quad (2)$$

where $\widetilde{V}(s') = V(s')$ if s' has already been explored, and otherwise $\widetilde{V}(s') = H(s')$, with $H : \mathcal{S} \rightarrow \mathbb{R}_+$. H is a heuristic function that initializes the value of unexplored states. It is proved (e.g. (Hansen and Zilberstein 2001)), that heuristic algorithms converge to an optimal solution iff H is admissible, i.e.: $\forall s \in \mathcal{S}, H(s) \leqslant V^{\pi^*}(s)$. The closer H is to V^{π^*}, the less states are explored, and the faster the algorithm converges. To be efficient in a domain-independent context, heuristic functions must be much easier to compute than the value function itself, and as close as possible to the optimal value function. To achieve these antagonist objectives, a good compromise consists in computing heuristic values on a relaxed planning domain.

[1] SSPs are often defined as goal-oriented MDPs with unit costs when not in a goal state, rather than any positive value like in our more general definition.

New STRIPS relaxation heuristics for SSPs

We here propose a generic extension of classical planning heuristics to SSPs, by reasoning about the "all-outcome determinization" of the MDP, generalizing the work by (Bonet and Geffner 2005). We show how to design admissible heuristics for SSPs from the deterministic case, and apply our theoretical extension to the h_{max} and h_{add} heuristics by (Bonet and Geffner 2001). Note that we could also have applied our extension to the h_{FF} heuristic (Hoffmann 2001).

As suggested by (Bonet and Geffner 2005), the `min-min` admissible heuristic $h_{m\text{-}m}$ is recursively defined for every reachable state $s \in \mathcal{S} \setminus \mathcal{G}$ by:

$$h_{m\text{-}m}(s) \leftarrow \min_{a \in app(s)} \left\{ C(s,a) + \min_{s' : T(s,a,s') > 0} h_{m\text{-}m}(s') \right\} \quad (3)$$

with the initial conditions: $h_{m\text{-}m}(s') = 0$ if $s' \in \mathcal{G}$ and $h_{m\text{-}m}(s') = +\infty$ otherwise. This heuristic counts the minimum number of steps required to reach a goal state in a non-deterministic relaxation of the domain. The `min-min` heuristic is well-informed but it naively searches in the original state space, so that it might explore as many states as non-heuristic algorithms. But clever heuristics that return a value lower or equal than $h_{m\text{-}m}$ still are admissible.

Let us give the intuition of the STRIPS relaxation heuristics by considering deterministic effects. As states are collections of atoms, only atoms added by successive actions need to be tracked down. As in (Bonet and Geffner 2001), we note $g_s(\omega)$ the cost of achieving an atom ω from a state s, i.e. the minimum number of steps required from s to have ω true. This value is computed by a forward chaining procedure where $g_s(\omega)$ is initially 0 if $\omega \in s$ and $+\infty$ otherwise:

$$g_s(\omega) \leftarrow \min_{\substack{a \in \mathcal{A} \text{ such that:} \\ \omega \in add(a)}} \{ g_s(\omega), cost(a) + g_s(prec(a)) \} \quad (4)$$

where $g_s(prec(a))$ denotes the cost of achieving the set of atoms in the preconditions of a. This requires to define the cost of achieving any set $\Delta \subseteq \Omega$ of atoms, what can be computed by aggregating the cost of each atom:

$$g_s(\Delta) = \bigoplus_{\omega \in \Delta} g_s(\omega) \quad (5)$$

When the fixed-point of eq. 4 is reached, the cost of achieving the set of goal states can be computed with eq. 5 and a heuristic value of s is $h_{\oplus}(s) = g_s(\mathcal{G})$. Two aggregation operators have been investigated by (Bonet and Geffner 2001): $\oplus = \max$ that gives rise to the h_{max} heuristic, such that $h_{max} \leqslant V^*$; $\oplus = \sum$ that provides the h_{add} heuristic, which is not admissible but often more informative.

Based on this helpful background, we can now extend h_{max} and h_{add} to the total criterion of the probabilistic case. For proof of admissibility, we are searching for a STRIPS relaxation heuristic whose value is lower than the `min-min` relaxation heuristic. Looking at eq. 3, this heuristic works

on a relaxed non-deterministic version of the original problem, known as the "all-outcome" determinization. This allows us to translate the search of a heuristic for the probabilistic problem into the search of a heuristic for a deterministic problem, as highlighted by the following proposition.

Proposition 1. *Let $\mathcal{M}$ be a goal-oriented MDP without reachable dead-ends. Let $\mathcal{D}$ be a deterministic planning problem (the "deterministic relaxation of $\mathcal{M}$"), obtained by replacing each probabilistic effect of each action of $\mathcal{M}$ by a deterministic action having the same precondition and the add and delete effects of the probabilistic effect. Then, an admissible heuristic for $\mathcal{D}$ is an admissible heuristic for $\mathcal{M}$.*

Proof. Let $app^{\mathcal{D}}$ be the applicability function for the deterministic problem. Eq. 3 reduces to $\mathrm{h}_{\text{m-m}}(s) \leftarrow \min_{a \in app^{\mathcal{D}}(s)} \{C(s,a) + \mathrm{h}_{\text{m-m}}(s')\}$, which is the update equation of optimal costs for $\mathcal{D}$. So the optimal cost of plans for $\mathcal{D}$ is equal to the value of the `min-min` relaxation for $\mathcal{M}$. Let h_{X} be an admissible heuristic for $\mathcal{D}$. h_{X} is lower than the optimal cost of plans for $\mathcal{D}$, i.e. than the value of $\mathrm{h}_{\text{m-m}}$ for $\mathcal{M}$, so it is admissible for $\mathcal{M}$. □

This proposition means that the $\mathrm{h}_{\max}$ heuristic computed in $\mathcal{D}$ is admissible for $\mathcal{M}$. It also demonstrates the admissibility of the heuristics used by `mGPT` (Bonet and Geffner 2005), but the `FF`-based one that is not admissible in $\mathcal{D}$.

However, such heuristics require to construct the deterministic relaxation of a goal-oriented MDP before solving it. To avoid this preliminary construction, we make this deterministic relaxation implicit in a new procedure that directly computes costs of atoms in the probabilistic domain:

$$g_s(\omega) \leftarrow \min_{\substack{a \in \mathcal{A} \text{ such that:} \\ \exists i \in [1; m_a], \omega \in add_i(a)}} \{g_s(\omega), cost(a) + g_s(prec(a))\} \quad (6)$$

Eq. 5 does not need to be changed for the probabilistic case. As in the deterministic case, we define the new $\mathrm{h}^+_{\max}$ for SSPs that is equal to $g_s(\mathcal{G})$ with the $\max$ aggregation operator.

Theorem 1. *The h^+_{max} heuristic is admissible for goal-oriented MDPs without reachable dead-ends.*

Proof. Let $\mathcal{M}$ be a goal-oriented MDP without reachable dead-ends and $\mathcal{D}$ be its deterministic relaxation. Eq. 6 for $\mathcal{M}$ boils down to eq. 4 for $\mathcal{D}$. Thus, $\mathrm{h}^+_{\max}$ in $\mathcal{M}$ has the same value as $\mathrm{h}_{\max}$ in $\mathcal{D}$. Since $\mathrm{h}_{\max}$ is admissible for $\mathcal{D}$, and using proposition 1, $\mathrm{h}_{\max}$ and thus $\mathrm{h}^+_{\max}$ are admissible in $\mathcal{M}$. □

We also define the new $\mathrm{h}^+_{\text{add}}$ for SSPs as $g_s(\mathcal{G})$ with the $\sum$ aggregation operator, but as in the deterministic case, $\mathrm{h}^+_{\text{add}}$ is not admissible. It is however more informative than $\mathrm{h}^+_{\max}$ and, as will be shown in the experiment section for the general discounted case, it is more efficient in practice.

Discounted Stochastic Shortest Path Problem

The previous $\mathrm{h}^+_{\max}$ and $\mathrm{h}^+_{\text{add}}$ heuristics, as well as heuristics by (Bonet and Geffner 2005), unfortunately are useless for goal-oriented MDPs where a policy execution may reach some dead-end state with a positive probability. As no goal state is reachable from a dead-end, $\mathrm{h}^+_{\max}$ and $\mathrm{h}^+_{\text{add}}$ may both return an infinite value for such state. Thus, because of eq. 2, the value of any preceding state will be infinite as well; after some iterations, this infinite value will propagate to at least one initial state. In fact, this fatal issue arises whatever the heuristic used: eq. 1 shows that, independently from heuristic values, the value of any dead-end state is equal to $+\infty$.

To cope with the divergence issue of the total criterion, we extend the previous SSP generalization of classical planning heuristics to *discounted* SSPs, which, like general MDP approaches, maximize the following *discounted* criterion:

$$\forall s \in \mathcal{S}, \pi^*(s) = \mathrm{argmin}_{\pi \in \mathcal{A}^{\mathcal{S}}} E\left[\sum_{t=0}^{+\infty} \gamma^t C_t \mid s_0 = s, \pi\right] \quad (7)$$

where $0 < \gamma < 1$ is a discount factor that ensures the convergence of the series of discounted costs for all classes of MDPs. In particular, values of all dead-ends are now finite and properly propagate to the initial state in the Bellman equation. For goal-oriented MDPs, this criterion allows us to define the following class of problems, that includes SSPs for $\gamma = 1$, and that has always a solution.

Definition 3. A *Discounted* Stochastic Shortest Path Problem (DSSP) is a goal-oriented MDP whose optimization criterion is given by eq. 7.

As highlighted in the introduction of this paper, note that DSSPs are different from the transformation of discounted MDP without termination states to an equivalent SSP done in (Bertsekas and Tsitsiklis 1996): in our case, we reason about goal states of the problem, whereas in the aforementioned book, either the MDP does not have any goal state or the SSP has at least one proper policy (i.e. does not have any dead-end). In this section, we present an extension of classical planning heuristics to goal-oriented MDPs with potential dead-ends, using the discounted criterion and targeting the true goal states, contrary to (Bertsekas and Tsitsiklis 1996). We prove the admissibility of the extended heuristics under some assumptions for the original heuristic, and apply this extension to the $\mathrm{h}_{\max}$ and h_{add} heuristics.

When reasoning on individual states, a well-informed heuristic for DSSPs can be obtained by inserting γ in eq. 3. Unfortunately, contrary to the non-discounted case, this "discounted" $\mathrm{h}_{\text{m-m}}$ heuristic does not generalize well to atom-space reasoning: eq. 6 that gave rise to our $\mathrm{h}^+_{\max}$ and $\mathrm{h}^+_{\text{add}}$ heuristics for SSPs, cannot be modified by simply inserting γ in the equation. The reason is that γ discounts *future* values, whereas eq. 6 is a forward procedure that updates *past* values. Naively inserting γ in this equation would lead to totally incoherent heuristic values.

For the sake of generality, we keep general positive costs in the definition of DSSPs, as did (Bonet and Geffner 2005) for instance in the case of SSPs. However, we caution the reader against using DSSPs with non-unit costs. Indeed, because of the discount factor and different transition costs, near dead-ends could become more interesting than far goal-states; thus, optimal policies could lead to dead-ends with a higher probability than to goal states. Yet, traditional approaches using the total criterion do not provide a better model, because they cannot solve the problem if there are reachable dead-ends. A more sophisticated approach, relying on bi-optimization of goal reachability probability and

costs of only paths reaching the goal, has been very recently proposed (Teichteil-Königsbuch 2012). This approach provides new theoretical foundations as well as some simple algorithmic means, to solve goal-oriented MDPs with reachable dead-ends and general costs (unit or non-unit, positive or negative). However, it has not been yet applied in a heuristic search context.

The generalization of our $\mathrm{h}^+_{\max}$ and $\mathrm{h}^+_{\mathrm{add}}$ heuristics to the discounted case relies on the computation of a lower bound on the minimum non-zero transition cost received along all paths starting from a state s, by means of a procedure inspired by eq. 6, that also reasons on individual atoms. This lower bound is required in our approach to handle general non-unit costs, but it is simply 1 in the unit-costs case. We compute an admissible heuristic for DSSPs by discounting successive steps and lowering all transition costs with this bound. To this purpose, we state and prove the following theorem, that is valid for general DSSPs, based on lower bounds on the minimum cost received along paths and the minimum number of steps required to reach a goal state.

Theorem 2. *Let s be a non-goal state of a DSSP, $c_s > 0$ be a lower bound on the minimum over all non-zero transition costs received from s by applying any policy, and $d_s > 1$ a lower bound on the number of steps required to reach a goal state from s by applying any policy ($d_s = +\infty$ if s is a dead-end). Then, the h^γ function defined as follows is an admissible heuristic:*

$$h^\gamma(s) = \left\{ \begin{array}{ll} c_s \sum_{t=0}^{d_s-1} \gamma^t & \textit{if } d_s < +\infty \\ c_s/(1-\gamma) & \textit{otherwise} \end{array} \right\} = c_s \frac{1-\gamma^{d_s}}{1-\gamma}$$

Proof. Let $\Phi^{\pi^*}(s)$ be the infinite but countable set of execution paths of π^* starting in s. Let $P(\phi)$ and $c(\phi)$ be resp. the probability and the (accumulated) cost of a path $\phi \in \Phi^{\pi^*}(s)$. Let $d(\phi)$ be the length of a path ϕ until it reaches a goal state ($d(\phi) = +\infty$ if ϕ does not reach a goal state). By definition of goal-oriented MDPs, all costs received after a goal is reached are equal to zero. By noting C_t^ϕ the cost received at time t along a path ϕ, we have: $c(\phi) = \sum_{t=0}^{+\infty} \gamma^t C_t^\phi = \sum_{t=0}^{d(\phi)-1} \gamma^t C_t^\phi \geqslant c_s \sum_{t=0}^{d_s-1} \gamma^t$ because $d(\phi) \geqslant d_s$ and $C_t^\phi \geqslant c_s$. Thus: $V^{\pi^*}(s) = \sum_{\phi \in \Phi^{\pi^*}(s)} P(\phi) c(\phi) \geqslant \sum_{\phi \in \Phi^{\pi^*}(s)} P(\phi) \left(c_s \sum_{t=0}^{d_s-1} \gamma^t \right) = c_s \sum_{t=0}^{d_s-1} \gamma^t$ because $\sum_{\phi \in \Phi^{\pi^*}(s)} P(\phi) = 1$. In the special case $d_s = +\infty$ (i.e. s is a dead-end), $V^{\pi^*}(s) \geqslant c_s \sum_{t=0}^{+\infty} \gamma^t = c_s/(1-\gamma)$. □

The previous theorem provides a new admissible heuristic for all discounted goal-oriented MDPs. In the next section, we will propose atom-space procedures to efficiently compute the lower bounds c_s and d_s.

New STRIPS relaxation heuristics for DSSPs

Atom-space computation of c_s. We can reuse the idea of the $\mathrm{h}_{\max}$ and $\mathrm{h}_{\mathrm{add}}$ heuristics, consisting in forgetting delete effects of the STRIPS domain. We collect all non-zero costs received by successively applying all applicable actions in the relaxed domain and keep the minimum one. Let $s \in \mathcal{S}$ be a state, $\omega \in \Omega$ be an atom and $c_s(\omega)$ be the minimum non-zero transition cost received until ω is added, starting from s. This value is computed by a forward chaining procedure where $c_s(\omega)$ is initially 0 if $\omega \in s$ and $+\infty$ otherwise:

$$c_s(\omega) \leftarrow \min_{\substack{a \in \mathcal{A} \text{ such that:} \\ \exists i \in [1; m_a], \omega \in add_i(a) \\ cost(a) > 0, c_s(prec(a)) < +\infty}} \left\{ c_s(\omega), cost(a), \overline{c_s(prec(a))} \right\} \quad (8)$$

where, for all set of atoms $\Delta \subseteq \Omega$, $c_s(\Delta)$ is the minimum cost received to add all atoms in Δ:

$$c_s(\Delta) = \min_{\omega \in \Delta} c_s(\omega) \quad (9)$$

When the fixed point of update equation 8 is reached, a lower bound on the minimum non-zero transition cost received along all paths starting from a state s is: $\mathrm{c_m}(s) = c_s(\Omega)$. For initialization purposes, we define a filter $\overline{c_s(prec(a))}$ as $cost(a)$ if $prec(a) \subseteq s$ and as $c_s(prec(a))$ otherwise.

Atom-space computation of d_s. d_s can be obtained by computing the $\mathrm{h}^+_{\max}$ heuristic on a slight variation of the input domain, where all non-zero costs are set to 1. Indeed, in such a case, $g_s(\omega)$ represents the minimum number of steps required to add the atom ω. Thus, $\mathrm{h}^+_{\max}(s) = \max_{\omega \in \mathcal{G}} g_s(\omega)$ is lower than the minimum number of steps required so that all atoms in $\mathcal{G}$ have been added, what is the intended lower bound. This simple observation combined with Theorem 2 allows us to derive the admissible heuristic $\mathrm{h}^\gamma_{\max}$ for DSSPs from $\mathrm{h}^+_{\max}$, the latter being computed by assuming that all non-zero action costs are equal to 1. We also derive a non-admissible but well-informed heuristic $\mathrm{h}^\gamma_{\mathrm{add}}$ from $\mathrm{h}^+_{\mathrm{add}}$. We note $\mathrm{h}^{1,+}_{\max}$ and $\mathrm{h}^{1,+}_{\mathrm{add}}$ the resp. $\mathrm{h}^+_{\max}$ and $\mathrm{h}^+_{\mathrm{add}}$ heuristics obtained when all non-zero action costs in eq. 6 are replaced by 1.

Definition 4. Let $\mathrm{c_m}(s)$ be a lower bound on the cost received from a state s by applying any policy as defined above, and $\mathrm{h}^{1,+}_{\mathrm{X}}$ a heuristic with values in $\mathbb{N}_+$ such that $\mathrm{h}^{1,+}_{\mathrm{X}}(s) = 0$ if s is a goal and $\mathrm{h}^{1,+}_{\mathrm{X}}(s) = +\infty$ if s is a dead-end. The $\mathrm{h}^\gamma_{\mathrm{X}}$ heuristic is defined as:

$$\mathrm{h}^\gamma_{\mathrm{X}}(s) = \mathrm{c_m}(s) \frac{1-\gamma^{\mathrm{h}^{1,+}_{\mathrm{X}}(s)}}{1-\gamma}$$

The $\mathrm{h}^{1,+}_{\max}$ and $\mathrm{h}^{1,+}_{\mathrm{add}}$ both satisfy above conditions, so that we can define $\mathrm{h}^\gamma_{\max}$ and $\mathrm{h}^\gamma_{\mathrm{add}}$. Note that the heuristics of (Bonet and Geffner 2005), that work only for goal-oriented MDPs without reachable dead-ends, can be generalized to the discounted case in the same way, so become helpful in presence of dead-ends. Moreover, with unit costs ($\mathrm{c_m}(s) = 1$ if s is not a goal state) and no dead-ends (we can safely set γ to 1), $\mathrm{h}^\gamma_{\mathrm{X}}$ reduces to $\mathrm{h}^+_{\mathrm{X}}$.

Finally, we prove that the $\mathrm{h}^\gamma_{\max}$ heuristic is admissible for all goal-oriented MDPs, with or without reachable dead-ends.

Theorem 3. *The h^γ_{max} heuristic is admissible for all goal-oriented MDPs (with or without dead-ends).*

Proof. Directly follows from Theorems 1 and 2. □

Another nice result is that $h^1_{max} = h^+_{max}$ and $h^1_{add} = h^+_{add}$: it means that h^γ_{max} and h^γ_{add} generalize h^+_{max} and h^+_{add} for DSSPs and for SSPs as well, rather than simply deriving them to a different use case.

To our knowledge, the new h^γ_{max} heuristic, generalized from classical planning heuristics, is an original atom-space *admissible* heuristic for all goal-oriented MDPs with or without dead-ends. Its complexity, like the original h_{max}, is *polynomial in the number of atoms*.

Experimental evaluations

Testbed description

We tested our approach on 150 problems from the fully observable probabilistic track of the 2008 International Planning Competition (IPC). Each problem was solved by 5 MDP heuristic algorithms (detailed below) with our h^γ_{max} and h^γ_{add} heuristics for each of them, and by previous winners of the past IPCs using their last available versions from their websites and optimized settings (such as 'all-outcome' effects for determinization-based planners and the ATLAS library): `FF-REPLAN` (Yoon, Fern, and Givan 2007), `FPG` (Buffet and Aberdeen 2009) and `RFF` (Teichteil-Königsbuch, Kuter, and Infantes 2010). Each problem was solved by $6 \times 2 + 3 = 15$ planners; in order to evaluate all planners on all problems in a reasonable time, each planner was given at most 10 minutes and 1.8 GB of RAM per problem. The machine used was an Intel Xeon running at 2.93 GHz in 64 bits mode on Ubuntu Linux.

We used the `mdpsim` simulator from IPCs to evaluate different criteria. As soon as a planner converges or exceeds its time or memory limits, the simulator executes the planner's policy 100 times starting from the problem's initial state. At each step of an execution, the simulator samples an outcome among the effects of the action specified by the planner's policy in the current state. Each run stops when a goal state is reached or after 1000 steps. Each IPC problem was solved as SSP with unit costs, and three relevant criteria were evaluated: **% Goal**: percentage of runs where the policy reaches a goal state; **Length (avg)**: average length of execution runs *that reach a goal state*; **Time**: total time (in seconds) to optimize the policy and execute the runs *that reach a goal state*. A problem is considered *solved* by a planner if its policy reaches a goal state with a non zero probability.

Candidate MDP heuristic algorithms.

We plugged h^γ_{max} and h^γ_{add} in several heuristic algorithms: `Improved-LAO*` (Hansen and Zilberstein 2001), `LRTDP` (Bonet and Geffner 2003), `LDFS` (Bonet and Geffner 2006), `sLAO*` (Feng and Hansen 2002), and `sRTDP` (Feng, Hansen, and Zilberstein 2003). For all planners and problems, the discount factor was $\gamma = 0.9$ and the precision $\epsilon = 0.001$. Of course, some problems without reachable dead-ends could have been solved with $\gamma = 1$ and achieve better '% goal' end 'length' criteria; but as we want to be domain-independent and solve all problems without analyzing them beforehand to detect possible reachable dead-ends (as in `exploding blocksworld`), we set the same $\gamma = 0.9 < 1$ for all problems.

Experimental results

We aggregated the results into three distinct categories for better readability: IPC winners, h^γ_{max} and h^γ_{add}. For each problem, the value of a given criterion for a given category is the best value of the criterion over all algorithms of the category. We first compare the overall performance of categories on a domain-by-domain basis, then we analyze performances of individual algorithms inside each category.

We first present "point clouds" plots that compare, for each criterion, all algorithms in a given category (e.g. IPC winners) against all algorithms of another category. Such plots exhibit a global view of categories performances without any aggregation (such as averages). Each point corresponds to one problem and two algorithms: it gives the criterion value of one algorithm in the x-axis category against one algorithm in the y-axis category for the same problem. (each problem appears as many times as the number of combinations of algorithms from both categories).

Comparison with winners of the past IPCs. The first row of Figure 1 compares all algorithms in the h^γ_{max} and h^γ_{add} categories, against all algorithms in the 'IPC winners' category. The '% goal' criterion shows that MDP heuristic algorithms with h^γ_{max} or h^γ_{add} globally reach the goal with a higher frequency than overall IPC winners for the problems which they could solve. Another interesting result is that `exploding blocksworld` problems do contain reachable dead-ends, and they are now solved by goal-oriented MDP heuristic approaches — what was not the case of `mGPT` (Bonet and Geffner 2005) nor `sLAO*` in the 2004 competition (Younes et al. 2005) — and better than IPC winners. The same holds for the `triangle-tireworld` problems. Moreover, for the problems that could be solved, the average lengths of h^γ_{max} and h^γ_{add} algorithms are often comparable to those of IPC winners, except for some domains where visible differences appear: for instance in `rectangle-tireworld`, our heuristics outperform IPC winners; this is not the case in `zenotravel`. Finally, as we expected, IPC winners slightly outperform h^γ_{max} or h^γ_{add} algorithms in terms of total time.

h^γ_{max} vs. h^γ_{add} comparisons. The second row of Figure 1 compares all MDP heuristic algorithms using the h^γ_{max} heuristic, against the same algorithms using the h^γ_{add} heuristic. Remind that h^γ_{max} is admissible but not h^γ_{add}: yet, strangely enough, the '% goal' criterion plot shows that h^γ_{add} achieves slightly better performances for this criterion for the `exploding blocksworld` and `triangle-tireworld` domains. The reason is that h^γ_{max} is less informative than h^γ_{add}: heuristic algorithms using h^γ_{max} could not converge within the 10 minutes limit for this domain (contrary to the same algorithms using h^γ_{add}), so that they produced worse suboptimal policies. The 'time' criterion plot also confirms that h^γ_{max} often consumes more time resources (including optimization time) to reach the goal than h^γ_{add}. Finally, the 'length' plot shows that h^γ_{add} often achieves better performance than h^γ_{max} for this criterion.

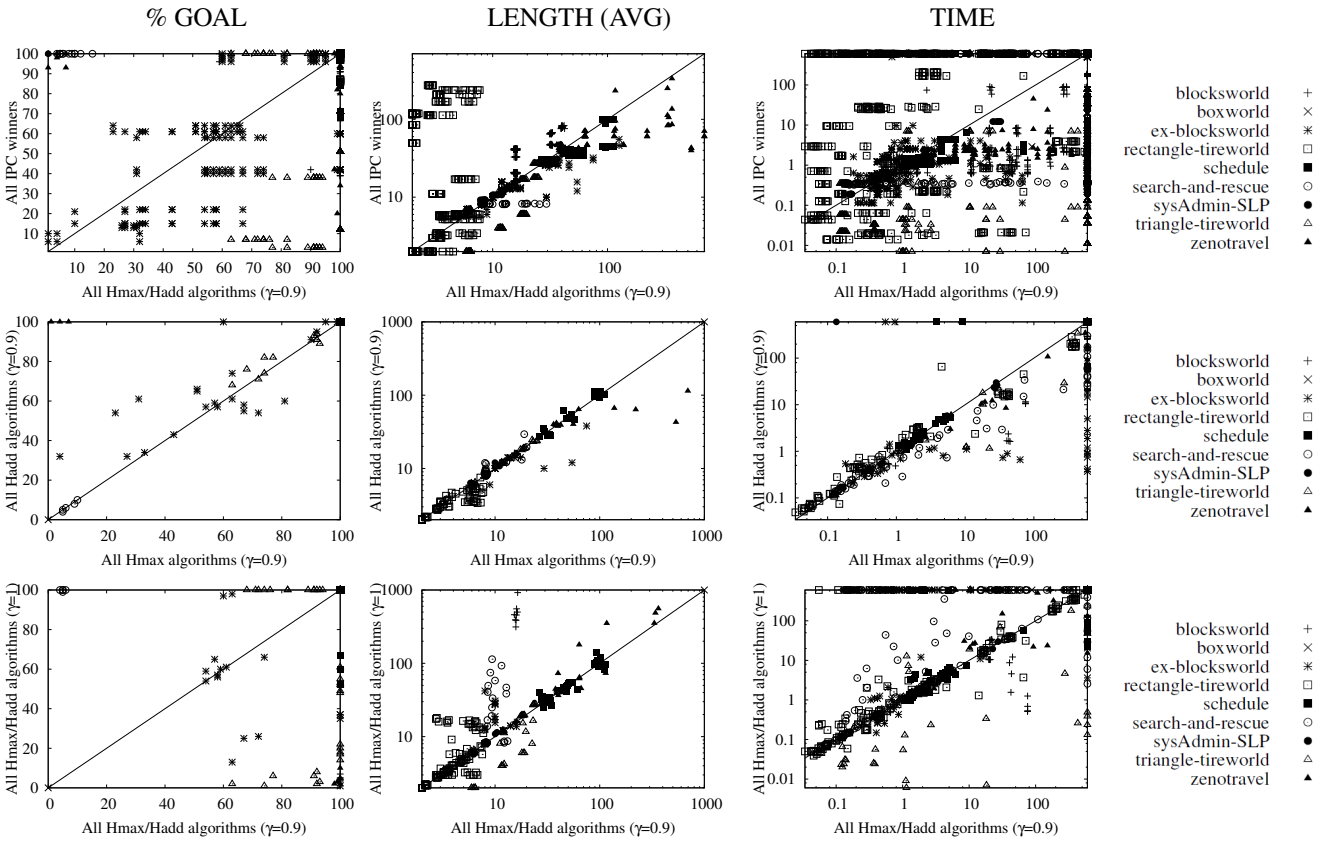


Figure 1: Face-to-face comparisons on problems of IPC 2008. 1st row: all $h^{\gamma=0.9}_{max}$ and $h^{\gamma=0.9}_{add}$ algorithms vs. all IPC winners. 2nd row: all $h^{\gamma=0.9}_{max}$ algorithms vs. all $h^{\gamma=0.9}_{add}$ algorithms. 3rd row: all $h^{\gamma=0.9}_{max}$ and $h^{\gamma=0.9}_{add}$ algorithms vs. all $h^{\gamma=1}_{max}$ and $h^{\gamma=1}_{add}$ algorithms.

$h^{0.9}_{max}$/$h^{0.9}_{add}$ vs. h^{1}_{max}/h^{1}_{add} comparisons. The third row of Figure 1 compares all MDP heuristic algorithms using the h^{γ}_{max} and h^{γ}_{add} heuristics with $\gamma = 0.9$, against themselves with $\gamma = 1$ (h^{+}_{max} and h^{+}_{add}). The '% Goal' criterion shows that discounted heuristics reach the goal more frequently than undiscounted ones in most domains except `search-and-rescue`, where the 0.9 discount factor is too small and prevents the planner from seeing the goal, and `triangle-tireworld`, where the best among $\gamma = 0.9$ and $\gamma = 1$ depends on the heuristic used. In all other domains, undiscounted heuristics return an infinite value so that the final policy is almost random and reaches the goal with a low probability (a proper policy with $\gamma = 1$ outperforms discounted policies). The 'length' criterion shows that discounted heuristics reach the goal with significantly less steps than undiscounted ones, suggesting that the latter give rise to almost random policies for most domains. Finally, the 'time' criterion clearly points out that discounted heuristics solve far more problems than undiscounted ones, what highlights the relevance of solving DSSPs as we propose, instead of SSPs as usual in the literature. Again, the SSP transformation by (Bertsekas and Tsitsiklis 1996) does not apply, because the goal of the transformed problem does not correspond to actual goals of the original problem.

Detailed comparisons. Table 1 presents more detailed algorithm performances in each category: IPC winners, h^{γ}_{max} and h^{γ}_{add}. The second column represents the number of problems solved by a given algorithm (i.e. it returns a policy reaching the goal with a positive probability). Other columns give for each criterion the number of problems where a given algorithm is the best among all algorithms that solved the same problems, over all categories or over its category (in addition to the average of the criterion over all problems for this algorithm): the highest the better. This table shows four main results: (1) all MDP heuristic algorithms achieve comparable performances on all criteria, meaning that h^{γ}_{max} and h^{γ}_{add} are not really impacted by the encoding of the policy graph nor by the search ordering of the algorithm; (2) h^{γ}_{add} performs slightly better than h^{γ}_{max} on all problems (it is not admissible but well-informed); (3) heuristic MDP algorithms using h^{γ}_{max} or h^{γ}_{add} globally achieve better performances than all IPC winners for the '% goal' and 'length' criteria; (4) heuristic MDP algorithms using h^{γ}_{max} or h^{γ}_{add} globally solve as many problems as `FF-REPLAN` and `FPG`, and outperform them on all other criteria.

Conclusion

We provide new atom-space heuristics for probabilistic planning with dead-ends generalized from efficient classical

Table 1: Comparisons over all problems of IPC 2008 between algorithms in the three categories: IPC winners, h^{γ}_{max}, and h^{γ}_{add} ($\gamma = 0.9$); subcategories are: IPC winners without RFF, and best of h^{γ}_{max}/h^{γ}_{add}. For each criterion: a **bold** number indicates the best element in overall or in its category, a framed number indicates that the overall category is the best for this criterion, and a $\star$ symbol indicates the best overall category between h^{γ}_{max}/h^{γ}_{add} and FPG/FF-REPLAN. For columns 3 to 5: the format of cells is 'a *(b)* [c]', where 'a' (resp. 'b') is the number of problems where the corresponding algorithm is the best among the algorithms that solved this problem in *all* categories (resp. *its* category), and 'c' is the average of the criterion among all problems solved. (Sub)categories are compared among problems that were solved by at least one algorithm in each category.

Algorithm	# Problems solved	% Goal	Length (avg)	Time
RFF	**89**	**7** ***(16)*** [83.22]	**4** ***(13)*** [76.82]	**4** *(9)* [13.20]
FPG	44	6 *(11)* [83.56]	1 *(4)* [86.41]	0 *(0)* [44.10]
FF-REPLAN	48	4 *(4)* [51.60]	**4** *(12)* [42.50]	0 ***(10)*** [20.34]
FPG/FF-REPLAN	**73** $\star$	26	15	10
IPC winners	**100**	43	26	**30**

	h^{γ}_{max}	h^{γ}_{add}	h^{γ}_{max}	h^{γ}_{add}	h^{γ}_{max}	h^{γ}_{add}	h^{γ}_{max}	h^{γ}_{add}
Improved-LAO*	**55**	**68**	**7** *(24)* [91.74]	**7** *(26)* [94.66]	1 ***(12)*** [29.11]	2 ***(14)*** [26.21]	0 ***(15)*** [20.04]	2 ***(9)*** [36.50]
LRTDP	41	50	**7** ***(25)*** [95.14]	**7** *(26)* [94.69]	1 *(10)* [13.49]	1 *(13)* [16.95]	0 *(1)* [24.58]	0 *(4)* [29.69]
LDFS	38	52	**7** *(24)* [92.60]	**7** ***(27)*** [94.78]	1 *(11)* [20.97]	1 *(11)* [19.18]	1 *(4)* [26.83]	1 *(7)* [14.45]
sLAO*	32	33	**7** *(24)* [93.62]	**7** *(26)* [94.45]	2 *(10)* [15.28]	1 *(13)* [14.48]	0 *(1)* [72.46]	0 *(0)* [40.28]
sRTDP	33	36	6 *(23)* [81.18]	**7** ***(27)*** [78.19]	1 *(3)* [13.41]	2 *(7)* [14.70]	0 *(1)* [102.59]	0 *(0)* [88.71]
Sub. h^{γ}_{max}/h^{γ}_{add}	55	**70**	42 $\star$	**45** $\star$	23 $\star$	**24** $\star$	9	**11** $\star$
h^{γ}_{max}/h^{γ}_{add}	72		**46** $\star$		**35** $\star$		19 $\star$	

planning heuristics: h^{γ}_{max}, for which we prove its admissibility, and h^{γ}_{add}, which is not admissible but often more informative than the former. These heuristics allow MDP heuristic algorithms to work in presence (or not) of dead-ends, i.e. when all possible policies reach with a positive probability a state from where no goal state can be achieved. Experimental results show that implementing efficient atom-space heuristics at the core of MDP heuristic algorithms rather than in an underlying deterministic planner, achieves nearly competitive performances with winners of past international planning competitions in terms of number of problems solved and computation time, while providing better goal reaching probability and average length to the goal.

One could argue that our heuristics do not take probabilities into account. Yet, many state-space heuristics for goal-oriented MDPs like the min-min relaxation also forget probabilities in their computation: the reason is that probabilities of paths in the relaxed problem are often very different from the ones in the original probabilistic problem, leading to non-admissible or poorly-informed heuristics. However, In the future, we intend to design well-informed atom-space heuristics that incorporate probabilities in their computations. Other interesting future research involves the direct insertion of γ inside the update equations of h^{+}_{max} or h^{+}_{add}, as well as the extension of our heuristics to hybrid goal-oriented MDPs with discrete and continuous state variables.

References

Bertsekas, D. P., and Tsitsiklis, J. N. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

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

Bonet, B., and Geffner, H. 2003. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. ICAPS*.

Bonet, B., and Geffner, H. 2005. mGPT: A probabilistic planner based on heuristic search. *JAIR* 24:933–944.

Bonet, B., and Geffner, H. 2006. Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to mdps. In *Proc. ICAPS*.

Buffet, O., and Aberdeen, D. 2007. FF+FPG: Guiding a policy-gradient planner. In *Proc. ICAPS*, 42–48.

Buffet, O., and Aberdeen, D. 2009. The factored policy-gradient planner. *Artificial Intelligence* 173(5-6):722–747.

Feng, Z., and Hansen, E. A. 2002. Symbolic heuristic search for factored markov decision processes. In *Proc. AAAI*, 455–460.

Feng, Z.; Hansen, E. A.; and Zilberstein, S. 2003. Symbolic generalization for on-line planning. In *Proc. UAI*, 209–216.

Gazen, B. C., and Knoblock, C. A. 1997. Combining the expressivity of UCPOP with the efficiency of graphplan. In *Proc. ECP*, 221–233.

Hansen, E. A., and Zilberstein, S. 2001. LAO*: A heuristic search algorithm that finds solutions with loops. *AIJ* 129(1-2):35–62.

Hoffmann, J. 2001. FF: The fast-forward planning system. *AI Magazine* 22(3):57–62.

Joshi, S.; Kersting, K.; and Khardon, R. 2010. Self-taught decision theoretic planning with first order decision diagrams. In *Proc. ICAPS*, 89–96.

Kolobov, A.; Mausam; and Weld, D. S. 2010. Classical

planning in MDP heuristics: With a little help from generalization. In *Proc. ICAPS*, 97–104.

Teichteil-Königsbuch, F.; Kuter, U.; and Infantes, G. 2010. Incremental plan aggregation for generating policies in MDPs. In *Proc. AAMAS*, 1231–1238.

Teichteil-Königsbuch, F. 2012. Stochastic safest and shortest path problems. In *AAAI'12: Proceedings of the 26th AAAI conference on Artificial intelligence*. AAAI Press.

Wu, J.-H.; Kalyanam, R.; and Givan, R. 2008. Stochastic enforced hill-climbing. In *Proc. ICAPS*, 396–403.

Yoon, S. W.; Ruml, W.; Benton, J.; and Do, M. B. 2010. Improving determinization in hindsight for on-line probabilistic planning. In *Proc. ICAPS*, 209–217.

Yoon, S.; Fern, A.; and Givan, R. 2007. FF-Replan: A baseline for probabilistic planning. In *Proc. ICAPS*, 352–359.

Younes, H. L. S.; Littman, M. L.; Weissman, D.; and Asmuth, J. 2005. The first probabilistic track of the International Planning Competition. *JAIR* 24:851–887.

Multi-Agent A* for Parallel and Distributed Systems

Raz Nissim and **Ronen Brafman**
Ben-Gurion University of the Negev
Be'er Sheva, Israel
raznis,brafman@cs.bgu.ac.il

Abstract

Search is among the most fundamental techniques for problem solving, and A* is probably the best known heuristic search algorithm. In this paper we adapt A* to the multi-agent setting, focusing on multi-agent planning problems. We provide a simple formulation of multi-agent A*, with a parallel and distributed variant. Our algorithms exploit the structure of multi-agent problems to not only distribute the work efficiently among different agents, but also to remove symmetries and reduce the overall workload. Given a multi-agent planning problem in which agents are not tightly coupled, our parallel version of A* leads to super-linear speedup, solving benchmark problems that have not been solved before. In its distributed version, the algorithm ensures that private information is not shared among agents, yet computation is still efficient – sometimes even more than centralized search – despite the fact that each agent has access to partial information only.

Introduction

As interest in multi-agent (MA) systems grows, so does the need to provide tools for multi-agent problem solving. Search, in particular, is among the most fundamental techniques for problem solving, hence the importance of adapting search algorithms to the MA setting.

A* is probably the most celebrated heuristic search algorithm. Its good theoretical properties make it the favorite algorithm when searching for a provably optimal solution. The main contribution of this paper is MA-A*, a multi-agent formulation of A*. MA-A* attempts to make the most of the parallel nature of the system, i.e., the existence of multiple computing agents, while respecting its distributed nature, when relevant, i.e., the fact that some information is local to an agent, and cannot be shared. It is not a shallow parallelization or distribution of A*, as some successful parallel implementations of A* (Kishimoto, Fukunaga, and Botea 2009). Rather, it is structure-aware, using the distinction between local and globally relevant actions and propositions to focus the work of each agent, dividing both states and operators among the agents, and exploiting symmetries that arise from the multi-agent structure. Moreover, MA-A* reduces exactly to A* when there is a single agent, unlike existing multi-core search methods (Tu et al. 2009; Burns et al. 2009). MA-A* comes in two flavors, a parallel one and a distributed one, that differ only in the nature of the heuristic functions used.

To evaluate MA-A* we apply it to a number of multi-agent planning problems, comparing its performance to the best current optimal centralized planner and to the best (non-optimal) distributed planner. In the parallel case, we show super-linear speed-up, on problems in which agents are not tightly coupled. This stems from the fact that our algorithm is able to exploit the internal structure of the problem, and not only the added computational power. Using this variant, we were able to solve a number of planning problems that were so far beyond the reach of the best centralized optimal planners. In the distributed case, the agents are constrained to use only information that is directly accessible to them, i.e., information about their own operators and non-private aspects of the operators of other agents. Thus, this variant is truly distributed, and private information is not shared. In that setting, one would hope that the distributed algorithm would do not much worse than the centralized one (which has access to all information, but less computing power). Here, we see that the lack of global information is costly. Yet, even now, as long as the system is somewhat decoupled, the distributed algorithm can outperform the centralized one.

In developing and describing MA-A*, we focus on *multi-agent planning*, using planning terminology and evaluating the algorithm on multi-agent planning problems. We do this for two reasons. First, it is the domain of primary interest to us, and to a large community of researchers on planning and multi-agent planning. Second, there exists a very simple and elegant formulation of multi-agent planning by Brafman and Domshlak (Brafman and Domshlak 2008), which makes explicit important notions, and in particular that of private and public variables and actions. However, search spaces with their operators, and planning spaces with their actions are almost synonymous, and so this development could be easily cast in search terminology.

Background

As noted earlier, we focus on multi-agent search in the context of multi-agent planning. MA planning is a wide and well studied field of research (Durfee 2001; Jonsson and Rovatsos 2011; ter Mors et al. 2010), but the most appropriate framework for our work is the MA-STRIPS model (Brafman and Domshlak 2008), presented by Brafman and Domsh-

lak (BD, for short). This framework minimally extends the classical STRIPS problem to MA planning for cooperative agents. The benefit of using this minimalistic model is that it is easy to see how the well known relationship between (single-agent) planning and general search is maintained in the multi-agent case, and that insights obtained using it could apply to more complex models of multi-agent planning.

A MA-STRIPS problem for a set of agents $\Phi = \{\varphi_i\}_{i=1}^{k}$ is given by a 4-tuple $\Pi = \langle P, \{A_i\}_{i=1}^{k}, I, G\rangle$, where P is a finite set of propositions, $I \subseteq P$ and $G \subseteq P$ encode the initial state and goal, respectively, and for $1 \leq i \leq k$, A_i is the set of actions agent φ_i is capable of performing. Each action $a = \langle pre(a), eff(a)\rangle$ is given by its preconditions and effects.

The MA-STRIPS model distinguishes between private and public variables and operators. A *private* variable of agent φ is required and affected only by the actions of φ. An action is *private* if all variables it affects and requires are private. All other actions are classified as *public*. That is, φ's private actions affect and are affected only by φ, while its public actions may require or affect the actions of other agents. For ease of the presentation of MA-A* and for the brevity of its proof, we assume that all actions that achieve a goal condition are considered *public*. This assumption must hold in order to maintain completeness of MA-A* as presented, but the algorithm is easily modified to remove it.

Given a model of a distributed system such as MA-STRIPS, it is natural to ask how to search for a solution. The best known example of distributed search is that of distributed CSPs (Yokoo et al. 1998), and various search techniques and heuristics have been developed for it (Meisels 2007). Planning problems can be cast as CSP problems (given some bound on the number of actions), and the first attempt to solve MA-STRIPS problems was based on a reduction to distributed CSPs. More specifically, BD introduced the *Planning as CSP+Planning* methodology for planning by a system of cooperative agents with private information. This approach separates the public aspect of the problem, which involves finding public action sequences that satisfy a certain distributed CSP, from the private aspect, which ensures that each agent can actually execute these public actions in a sequence. Solutions found are locally optimal, in the sense that they minimize δ, the maximal number of public actions performed by an agent. This methodology was later extended to the first fully distributed MA algorithm for MA-STRIPS planning, *Planning-First* (Nissim, Brafman, and Domshlak 2010). *Planning First* was shown to be very efficient in solving problems where the agents are very loosely coupled, and where δ is very low. However, it does not scale up as δ rises, mostly due to the large search space of the Distributed CSP. As we will see later, the forward search algorithm we present scales much better *and* leads to a globally optimal solution.

We note that much work has been done attempting to solve MA planning in a partially observable and stochastic setting (using the DEC-POMDP model) (Szer, Charpillet, and Zilberstein 2005). This work, which uses different, more specialized methods and heuristics, falls out of the scope of this paper, which presents a *general* algorithm for solving the classical MA planning problem, using the MA-STRIPS formalism.

There is a growing interest in the use of parallelization techniques for scaling up planning algorithms. Recent results (Helmert and Röger 2008) show that in some cases, even almost perfect heuristics will not prevent exploring an exponential number of search nodes. Parallelization, as an orthogonal method of speeding up search, can continue the improvement of future planners. Kishimoto et al. (Kishimoto, Fukunaga, and Botea 2009) presented HDA*, a simple and effective parallelization of A*. HDA* distributes work between processors using a hash function, which assigns each state to a unique process. Communications are asynchronous and non-blocking, and duplicate detection is performed locally by each processor. HDA* was shown to achieve significant speedup using 4 processors, and to scale well up to 128 processors. Its efficiency (speedup divided by number of processors) ranges from 0.9 on 4-core machines, to 0.25 using 128 processors, constituting in sublinear speedup.

Multi-Agent A*

Overview

MA-A* is a distributed variation of A*, which maintains a separate search space for each agent. Each agent maintains an *open list* of states that are candidates for expansion and a *closed list* of already expanded states. It expands the state with the minimal $f = g + h$ value in its open list. When an agent expands state s, it uses its own operators only. This means that two agents expanding the same state will generate *different* successor states.

Since no agent has complete knowledge of the entire search space, messages must be sent, informing agents of open search nodes relevant to them. Agent φ_i characterizes state s as relevant to agent φ_j if φ_j has a public operator whose public preconditions (the preconditions φ_i is aware of) hold in s. In principle, a relevant state *must* be sent to φ_j (and this is what A* would effectively do). However, in some cases, this can be avoided, and there is also some flexibility as to when precisely the message will be sent. We discuss these finer details later, and for now, assume a relevant state is sent once it is generated.

The messages sent between agents contain the full state s, i.e. including both public and private variable values, as well as the cost of the best plan from the initial state to s found so far, and the sending agent's heuristic estimate of s[1]. When agent φ receives a state via a message, it checks whether this state exists in its open or closed lists. If it does not appear in these lists, it is inserted into the open list. If a copy of this state with higher g value exists in the open

[1]It may appear that agents are revealing their private data because they transmit their private state in their messages. However, as will be apparent in the algorithm, other agents do not use this information in any way, nor alter it. They simply copy it to future states. Only the agent itself can change the value of its private state. Consequently, this data can be encrypted arbitrarily – it is merely used as an ID by other agents.

list, its g value is updated, and if it is in the closed list, it is reopened. Otherwise, it is discarded. Whenever a received state is (re)inserted into the open list, the agent computes its local h_φ value for this state, and assigns the maximum of its h_φ value and the h value in the received message.

Once an agent expands a solution state s, it sends s to all agents and initiates the process of verifying its optimality. When the solution is verified as optimal, the agent initiates the trace-back of the solution plan. This is also a distributed process, which involves all agents that perform some action in the optimal plan. When the trace-back phase is done, a terminating message is broad-casted.

The MA-A* Algorithm

Algorithms 1-3 depict the MA-A* algorithm for agent φ_i.

Algorithm 1 MA-A* for Agent φ_i

```
1: while did not receive true from a solution verification procedure do
2:     for all messages m in message queue do
3:         process-message(m)
4:     s ← extract − min(openlist)
5:     expand(s)
```

Algorithm 2 process-message($m = \langle s, g_{\varphi_j}(s), h_{\varphi_j}(s)\rangle$)

```
1: if s is not in open or closed list or g_{φi}(s) > g_{φj}(s) then
2:     add s to open list and calculate h_{φi}(s)
3:     g_{φi}(s) ← g_{φj}(s)
4:     h_{φi}(s) ← max(h_{φi}(s), h_{φj}(s))
```

Algorithm 3 expand(s)

```
1: move s to closed list
2: if s is a goal state then
3:     broadcast s to all agents
4:     initiate verification of stable property f_{lower−bound} ≥ g_{φi}(s)
5:     return
6: for all agents φj ∈ Φ do
7:     if the last action leading to s was public and φj has a public action for which all public preconditions hold in s then
8:         send s to φj
9: apply φi's successor operator to s
10: for all successors s′ do
11:     update g_{φi}(s′) and calculate h_{φi}(s′)
12:     if s′ is not in closed list or f_{φi}(s′) is now smaller than it was when s′ was moved to closed list then
13:         move s′ to open list
```

The pseudo-code is mostly self-explanatory, but some of its finer issues require further discussion.

Termination Detection Unlike in A*, expansion of a goal state in MA-A* does not necessarily mean an optimal solution has been found. In our case, a solution is known to be optimal only if all agents prove it so. Intuitively, a solution state s having solution cost f^* is known to be optimal if there exists no state s' in the open list or the input channel of some agent, such that $f(s') < f^*$. In other words, solution state s is known to be optimal if $f(s) \leq f_{lower-bound}$, where $f_{lower-bound}$ is a lower bound on the f-value of the entire system (which includes all states in all open lists, as well as states in messages that have not been processed, yet).

To detect this situation, we use Chandy and Lamport's *snapshot algorithm* (Chandy and Lamport 1985), which enables a process to create an approximation of the global state of the system, without "freezing" the distributed computation. Although there is no guarantee that the computed global state actually occurred, the approximation is good enough to determine whether a stable property currently holds in the system. A property of the system is *stable* if it is a global predicate which remains true once it becomes true. Specifically, properties of the form $f_{lower-bound} \geq c$ for some fixed value c, are stable when h is a *globally consistent* heuristic function. That is, when f values cannot decrease along a path. In our case, this path may involve a number of agents, each with its h values. If each of the local functions h_φ are consistent, and agents apply the $\max$ operator when receiving a message, as explained above, this property holds.

We note that for simplicity of the pseudo-code we omitted the detection of a situation where a goal state does not exist. This can be done by determining whether the stable property *"there are no open states in the system"* holds, using the same *snapshot algorithm*.

Parallel vs. Distributed Search via Heuristic Choice The quality of the heuristic function plays an important role in the success or failure of a forward search algorithm. In MA-A*, the nature of the heuristic is also the distinguishing factor between parallel and distributed search.

In centralized search, the global heuristic function is computed having *complete knowledge* of the problem. Specifically, it is aware of all operators in the system. Parallel search attempts to speed-up centralized search using multiple processors that have access to the complete problem description. We achieve this by allowing each agent complete knowledge of both private and public operators of all agents. Thus, all agents compute (and can share) the global heuristic function, meaning that $h_{\varphi_i}(s) = h_{\varphi_j}(s)$ for all agents $\varphi_i, \varphi_j \in \Phi$ and for all states s. We refer to this version of MA-A* as MAP-A*.

Existing techniques for parallel search distribute either the *workload* (e.g. HDA* (Kishimoto, Fukunaga, and Botea 2009)) or the *operators* (e.g. ODMP (Vrakas, Refanidis, and Vlahavas 2001)) between processors. These distribution methods are mostly independent of the problem structure (hash function for workload distribution, for example). In contrast, MAP-A* distributes the workload *and* search operators between agents according to the **MA structure** of the problem.

In the distributed setting, we assume that agents have access to public information and their own private information only. Because each agent has different information, it must compute its own local heuristic function. More specifically, each agent knows the complete description of its private and public operators. In addition, it is aware of the public aspects of all agents' public operators. Recall that any public action a of agent φ_i can have both public and private preconditions and effects. Agents other than φ_i have access to a projected version of a, containing only public preconditions and effects. Thus, each agent can compute its heuristic estimate using a domain description that contains its own actions, as well as all public actions projected to public variables. The algorithm is completely agnostic as to how the agent uses this description to compute its private heuristic function. This allows us great flexibility, since different agents may use different heuristics. In fact, this is the *essence* of distributed search – each agent is a separate entity, capable of making choices regarding how it performs search. We refer to this distributed version of MA-A* as MAD-A*.

Relevancy and Timing of the Messages State s is considered relevant to agent φ_j if it has a public action for which all public preconditions hold in s and the last action leading to s was public (line 7 of Alg. 3). This means that all states that are products of private actions are considered irrelevant to other agents. As it turns out, since private actions do not affect other agents' capability to perform actions, an agent must send only states in which the last action performed was public, in order to maintain completeness (and optimality, as proved in the next section). Regarding states that are products of private actions as irrelevant, decreases communication, while effectively pruning large, symmetrical parts of the search space. The exploitation of symmetry is discussed in further detail later on.

As was hinted earlier, there exists some flexibility regarding when these relevant states are sent. A* can be viewed as essentially "sending" every state (i.e., inserting it to its open list) once it is generated. In MA-A*, relevant states can be sent when they are expanded (as in the pseudo-code) or once they are generated (changing Alg. 3 by moving the for-loop on line 6 inside the for-loop on line 10). The timing of the messages is especially important in the distributed setting, where agents may have different heuristic estimations. Sending the messages once they are generated increases communication, but allows for states that are not considered promising by some agent to be expanded by another agent in an earlier stage. Sending relevant states when they are expanded, on the other hand, decreases communication, but delays the sending of states not viewed as promising. Experimenting with the two options, we found that the lazy approach, of sending the messages only when they are expanded, dominates the other, most likely because communication is costly.

Proof of Optimality

First, we prove the following lemmas regarding the solution structure of a MA planning problem. We must note that as it is presented, MA-A* maintains completeness only if all actions which achieve some goal condition are considered public. This property is assumed throughout this section, but the algorithm is easily modified to remove it.

Lemma 1. *Let $P = (a_1, a_2 \ldots, a_k)$ be a legal plan for a MA planning problem Π. Let a_i, a_{i+1} be two consecutive actions taken in P by different agents, of which at least one is private. Then $P' = (a_1, \ldots, a_{i+1}, a_i, \ldots, a_k)$ is a legal plan for Π and $P(I) = P'(I)$.*

Proof. By definition of private and public actions, and because a_i, a_{i+1} are actions belonging to different agents, $varset(a_i) \cap varset(a_{i+1}) = \emptyset$, where $varset(a)$ is the set of variables which affect and are affected by a. Therefore, a_i does not achieve any of a_{i+1}'s preconditions, and a_{i+1} does not destroy any of a_i's preconditions. Therefore, if s is the state in which a_i is executed in P, a_{i+1} is executable in s, a_i is executable in $a_{i+1}(s)$, and $a_i(a_{i+1}(s)) = a_{i+1}(a_i(s))$. Therefore, $P' = (a_1, \ldots, a_{i+1}, a_i, \ldots, a_k)$ is a legal plan for Π. Since the suffix $(a_{i+2}, a_{i+3}, \ldots, a_k)$ remains unchanged in P', $P(I) = P'(I)$, completing the proof. □

Corollary 1. *For a MA planning problem Π for which an optimal plan $P = (a_1, a_2, \ldots, a_k)$ exists, there exists an optimal plan $P' = (a'_1, a'_2, \ldots, a'_k)$ for which the following restrictions apply:*

1. *If a_i is the first public action in P', then $a_1, \ldots, a_i$ belong to the same agent.*
2. *For each pair of consecutive public actions a_i, a_j in P', all actions a_l, $i < l \leq j$ belong to the same agent.*

Proof. Using repeated application of Lemma 1, we can move any ordered sequence of private actions performed by agent φ, so that it would be immediately before φ's subsequent public action and maintain legality of the plan. Since application of Lemma 1 does not change the cost of plan, the resulting plan is cost-optimal as well. □

Before proving the optimality of our algorithm, we prove the following lemma, which is a MA extension to a well known result for A*. In what follows, we have tacitly assumed a *liveness* property with the conditions that every sent message eventually arrives at its destination and that all agent operations take a finite amount of time. Also, for the clarity of the proof, we assume the atomicity of the *expand* and *process-message* procedures.

Lemma 2. *For any non-closed node s and for any optimal path P from I to s which follows the restrictions of Lemma 1, there exists an agent φ which either has an open node s' **or** has an incoming message containing s', such that s' is on P and $g_\varphi(s') = g^*(s')$.*

Proof. : Let $P = (I = n_0, n_1, \ldots, n_k = s)$. If I is in the open list of some agent φ (φ did not finish the algorithm's first iteration), let $s' = I$ and the lemma is trivially true since $g_\varphi(I) = g^*(I) = 0$. Suppose I is closed for all agents. Let Δ be the set of all nodes n_i in P that are closed by some agent φ, such that $g_\varphi(n_i) = g^*(n_i)$. Δ is not empty, since by assumption, $I \in \Delta$. Let n_j be the element of Δ with the highest index, closed by agent φ. Clearly, $n_j \neq s$ since s is non-closed. Let a be the action causing the transition $n_j \rightarrow n_{j+1}$ in P. Therefore, $g^*(n_{j+1}) = g_\varphi(n_j) + cost(a)$.

If φ is the agent performing a, then n_{j+1} is generated and moved to φ's open list in lines 9-13 of Algorithm 3, where $g_\varphi(n_{j+1})$ is assigned the value $g_\varphi(n_j) + cost(a) = g^*(n_{j+1})$ and the claim holds.

Otherwise, a is performed by agent $\varphi' \neq \varphi$. If a is a public action, then all its preconditions hold in n_j, and therefore n_j is sent to φ' by φ in line 8 in Algorithm 3. If a is a private action, by the definition of P, the next *public* action a' in P is performed by φ'. Since private actions do not change the values of public variables, the public preconditions of a' must hold in n_j, and therefore n_j is sent to φ' by φ in line 8 in Algorithm 3. Now, if the message containing n_j has been processed by φ', n_j has been added to the open list of φ' in Algorithm 2 and the claim holds since $g_{\varphi'}(n_j) = g_\varphi(n_j) = g^*(n_j)$. Otherwise, φ' has an incoming (unprocessed) message containing n_j and the claim holds since $g_\varphi(n_j) = g^*(n_j)$. □

Corollary 2. *Suppose h_φ is admissible for every $\varphi \in \Phi$, and suppose the algorithm has not terminated. Then, for any optimal solution path P which follows the restrictions of Lemma 1 from I to any goal node $s^\star$, there exists an agent φ_i which either has an open node s **or** has an incoming message containing s, such that s is on P **and** $f_{\varphi_i}(s) \leq h^*(I)$.*

Proof. : By Lemma 2, for every restricted optimal path P, there exists an agent φ_i which either has an open node s **or** has an incoming message containing s, such that s is on P and $g_{\varphi_i}(s) = g^*(s)$. By the definition of f, and since h_{φ_i} is admissible, we have in both cases:

$$f_{\varphi_i}(s) = g_{\varphi_i}(s) + h_{\varphi_i}(s) = g^*(s) + h_{\varphi_i}(s)$$
$$\leq g^*(s) + h^*(s) = f^*(s)$$

But since P is an optimal path, $f^*(n) = h^*(I)$, for all $n \in P$, which completes the proof. □

Another lemma must be proved regarding the solution verification process. We assume global consistency of all heuristic functions, since all admissible heuristics can be made consistent by locally using the pathmax equation (Méro 1984), and by using the *max* operator as in line 4 of Alg. 2 on heuristic values of different agents. This is required since we need $f_{lower-bound}$ to be monotonic non-decreasing.

Lemma 3. *Let φ be an agent which either has an open node s **or** has an incoming message containing s. Then, the solution verification procedure for state s^* with $f(s^*) > f_\varphi(s)$ will return false.*

Proof. Let φ be an agent which either has an open node s **or** has an incoming message containing s, such that $f_\varphi(s) < f(s^*)$ for some solution node s^*. The solution verification procedure for state s^* verifies the stable property $p =$ "$f(s^*) \leq f_{lower-bound}$". Since $f_{lower-bound}$ represents the lowest f-value of any open or unprocessed state in the system, we have $f_{lower-bound} \leq f_\varphi(s) < f(s^*)$, contradicting p. Relying on the correctness of the snapshot algorithm, this means that the solution verification procedure will return false, proving the claim. □

We can now prove the optimality of our algorithm.

Theorem 1. *Algorithm 1 terminates by finding a cost-optimal path to a goal node, if one exists.*

Proof. : We prove this theorem by assuming the contrary - the algorithm does not terminate by finding a cost-optimal path to a goal node. 3 cases are to be considered:

1. *The algorithm terminates at a non-goal node.* This contradicts the termination condition, since the solution verification procedure is initiated only when a goal state is expanded.
2. *The algorithm does not terminate.* Since we are dealing with a finite search space, let $\chi(\Pi)$ denote the number of possible *non-goal* states. Since there are only a finite number of paths from I to any node s in the search space, s can be *reopened* a finite number of times. Let $\rho(\Pi)$ be the maximum number of times any *non-goal* node s can be reopened by any agent. Let t be the time point when all non-goal nodes s with $f_\varphi(s) < h^*(I)$ have been closed forever by all agents φ. This t exists since *a*) we assume liveness of message passing and agent computations; *b*) after at most $\chi(\Pi) \times \rho(\Pi)$ expansions of non-goal nodes by φ, all non-goal nodes of the search space must be closed forever by φ; and *c*) no goal node s^* with $f(s^*) < h^*(I)$ exists[2].
   By Corollary 2 and since an optimal path from I to some goal state s^* exists, some agent φ expanded state s^* at time t', such that $f_\varphi(s^*) \leq h^*(I)$. Since s^* is an optimal solution, if $t' \geq t$, $f_{lower-bound} \geq f_\varphi(s^*)$ at time t'. Therefore, φ's verification procedure of s^* will return true, and the algorithm terminates.
   Otherwise, $t' < t$. Let φ' be the last agent to close a non-goal state s with $f_{\varphi'}(s) < f_\varphi(s^*)$. φ' has s^* in its open list or as an incoming message. This is true because s^* has been broad-casted to all agents by φ, and because every time s^* is closed by some agent (when it expands it), it is immediately broad-casted again, ending up in the agent's open list or in its message queue. Now, φ' has no more open nodes with f-value lower than s^*, so it will eventually expand s^*, initiating the solution verification procedure which will return true, since $f_{lower-bound} \geq f_\varphi(s^*)$. This contradicts the assumption of non-termination.
3. *The algorithm terminates at a goal node without achieving optimal cost.* Suppose the algorithm terminates at some goal node s with $f(s) > h^*(I)$. By Corollary 2, there existed just before termination an agent φ having an open node s', or having an incoming message containing s', such that s' is on an optimal path and $f_\varphi(s') \leq h^*(I)$. Therefore, by Lemma 3, the solution verification procedure for state s will return false, contradicting the assumption that the algorithm terminated.

This concludes the proof. □

[2]This is needed since the number of *goal* node expansions is not bounded.

MA Planning Framework

One of the main goals of this work is to provide a general framework for solving the MA planning problem. We believe that such a framework will provide researchers with fertile ground for developing new search techniques and heuristics for MA planning.

We chose Fast Downward (Helmert 2006) (FD) as the basis for our MA framework – **MA-FD**. FD is currently the leading framework for planning, both in the number of algorithms and heuristics that it provides, and in terms of performance – winners of the past three international planning competitions were implemented on top of it. FD is also well documented and supported, so implementing and testing new ideas is relatively easy.

MA-FD uses FD's translator and preprocessor, with minor changes to support the distribution of operators to agents. In addition to the PDDL files describing the domain and the problem instance, MA-FD receives a file detailing the number of agents, their names, and their IP addresses. The agents do not have shared memory, and all information is relayed between agents using messages. Inter-agent communication is performed using the TCP/IP protocol, which enables running multiple MA-FD agents as processes on multi-core systems, networked computers/robots, or even the cloud. MA-FD is therefore fit to run as is on *any* number of (networked) processors, in both its parallel and distributed setting.

Both MAP-A* and MAD-A* are currently implemented, and since both settings have full flexibility regarding the heuristics used by agents, all admissible heuristics available on FD are also available on MA-FD. New heuristics are easily implementable, as in FD, and creating new search algorithms can also be done with minimal effort, since MA-FD provides the ground-work (parsing, communication, etc.).

Empirical results

In order to evaluate MA-A*, in both its parallel and distributed setting, we performed experiments on IPC (ICAPS) benchmarks, in domains where tasks can be naturally casted as MA problems. The *Satellites* and *Rovers* domains where motivated by real MA applications used by NASA. *Satellites* requires planning and scheduling observation tasks between multiple satellites, each equipped with different imaging tools. *Rovers* involves multiple rovers navigating a planetary surface, finding samples and communicating them back to a Lander. *Logistics* and *Zenotravel* are two transportation domains, where multiple vehicles transport packages or people to their destination.

For each planning problem, we ran two configurations of centralized A*, one using the LM-cut heuristic (Helmert and Domshlak 2009) and the other using the Merge&Shrink heuristic[3] (Helmert, Haslum, and Hoffmann 2007), as well as 4 configurations of MA-A*: MAP-A* and MAD-A* using LM-cut[4] and Merge&Shrink, running on multiple processors[5]. All experiments were run on a AMD Phenom 9550 2.2GHZ quad-core processor. Time limit was set at 1.5 hours, and memory usage was limited to 4GB, regardless of the number of cores used.

We begin by discussing performance of MA-A* in its parallel setting. Table 1 depicts the runtimes, number of expanded nodes and efficiency values (speedup divided by the number of processors). In Rovers and Satellites, domains in which the agents are loosely-coupled, MAP-A* overwhelmingly dominates centralized A*. MAP-A* outperformed A* in all but one very small instance of Rovers, solving 6 problems unsolved by A*. Running on multiple processors, MAP-A* exhibited super-linear speedup, showing efficiency ranging between 1 to 19.5 in problems solved by both planners. This result is unique because existing parallel planners exhibit sublinear speedup, bounding their efficiency to be ≤ 1. In fact, super-linear speedup suggests that MA-A* exploits the *structure* of the problem and not only the additional computational power. This is discussed in further detail in the next section. In Logistics and Zenotravel, where the agents are tightly-coupled, MAP-A*'s efficiency decreases. Using LM-cut, efficiency values range from 0.46 to 2, but even though efficiency decreased in these tightly-coupled systems, MAP-A* running on multiple processors still always outperforms its centralized counterpart. MAP-A* using Merge&Shrink does not perform well in these tightly-coupled domains. In $Zenotravel10$, for example, efficiency drops as low as 0.15, causing a $\times 2.2$ speed-down. In tightly coupled systems, many states are sent as messages, because many of the actions are public. Communication is more time consuming than local computations, causing a drop in efficiency. The situation becomes more acute when using Merge&Shrink, which in general is less accurate (but faster to compute) than LM-cut, because as more states are expanded, more are sent between agents.

Table 2 shows the running time and the average of the agents' initial state h-values of centralized A* and MAD-A* running on multiple processors, as well as running time and δ-values of Planning First, running on a single processor. MAD-A* performed very well in comparison to (non-optimal) Planning First, outperforming it in all but one task, and solving 6 more problems, while achieving optimality[6]. When comparing MAD-A* to centralized A*, our intuition is that efficiency will decrease, due to the inaccuracy of the heuristic estimates. In fact, the local heuristics of the agents are much less accurate than the global heuristics, as is apparent from the lower average h values of the initial state, and by the higher number of states expanded by MAD-A*. In the tightly-coupled domains, we do notice very low (sublinear) efficiency values. However, in Satellites and Rovers, MAD-A* using Merge&Shrink exhibits nearly linear and super-linear speedup, respectively, solving 2 problems not solved by centralized A*. Our intuition is that the

[3] We used the LFPA configuration of Merge&Shrink with abstraction size limit = 50K.

[4] LM-cut is not consistent, so in order to maintain completeness, we enforced local consistency using the pathmax equation.

[5] In problems having up to 4 agents, each agent was allocated a processor. For problems with 5 agents, one processor was shared by two agents.

[6] Efficiency of MAD-A* w.r.t Planning First is not shown, but is super-linear except in $Rovers5$ using LM-cut.

Table 1: Comparison of centralized A* and MAP-A* running on multiple processors. Running time (in sec.), number of expanded nodes and efficiency values are shown.

		LM-cut heuristic					Merge&Shrink heuristic				
Problem	#	A*		MAP-A*			A*		MAP-A*		
(# of processors)	agents	time	expands	time	expands	eff.	time	expands	time	expands	eff.
logistics7-1(4)	4	55.3	155289	27	504102	0.51	0	1624	0.8	36847	0
logistics8-1(4)	4	24.1	43665	13	168545	0.46	0.1	45	0.8	3552	0.03
logistics10-0(4)	5	203	193846	66.6	627314	0.76	81.7	3219478	75.9	3056185	0.27
Rovers3(2)	2	0	50	0	90	1.00	0	12	0	98	1.00
Rovers4(2)	2	0	9	0.04	68	0	0	9	0.08	123	0
Rovers5(2)	2	8.8	37397	1.8	18975	2.44	11.7	419110	5.8	266433	1.01
Rovers6(2)	2	–	–	236	2255393	∞	–	–	238	17402585	∞
Rovers7(3)	3	6.7	18315	1	12929	2.23	55.9	2890357	9.1	762620	2.05
Rovers8(4)	4	–	–	154	1271971	∞	–	–	–	–	N/A
Rovers12(4)	4	12.1	15222	0.9	10704	3.36	119	3577293	6.6	456744	4.51
Rovers14(4)	4	–	–	598	5311741	∞	–	–	–	–	N/A
satellites5(3)	3	1.3	1174	0.1	793	4.33	7	117756	2.2	82579	1.06
satellites6(3)	3	3.5	2976	0.2	1650	5.83	23.5	495348	0.4	19062	19.58
satellites7(4)	4	94.5	36652	12.4	53465	1.91	–	–	347	17042327	∞
satellites8(4)	4	–	–	94.8	345667	∞	–	–	–	–	N/A
satellites9(4)	5	–	–	105	2132756	∞	–	–	–	–	N/A
satellites10(4)	5	–	–	61.8	95192	∞	–	–	–	–	N/A
zenotravel9(3)	3	72.1	15408	20	29321	1.20	56.7	1590010	81.7	1526996	0.23
zenotravel10(3)	3	16.1	1587	4.3	3453	1.25	26.7	388274	60.5	770787	0.15
zenotravel12(3)	3	458	41311	57	41819	2.68	–	–	–	–	N/A
zenotravel13(3)	3	–	–	382	185827	∞	–	–	–	–	N/A

same properties that helped MAP-A* achieve super-linear speedup, kept MAD-A*, with its less accurate heuristics, on par with centralized search in loosely-coupled domains.

Discussion

MA-A* raises a number of research challenges and opportunities, which we now discuss. Naturally, the first question is why it works well in weakly coupled environments. It is known that when using a consistent heuristic, A* is optimal in the number of nodes it expands to recognize an optimal solution. In principle, it appears that MA-A* expands the same search tree, so it is not clear, a-priori, why we reach super-linear speedup. We believe there are two reasons for this: one is delayed expansion of some nodes, and the other is symmetry exploitation.

Delayed expansion refers to the fact that in MA-A*, when an agent expands a node, it does not apply all actions to it, but only its actions. This means that nodes are expanded in parts. This is somewhat analogous to pruning methods used in planning, known as helpful actions (Hoffmann and Nebel 2001). The idea behind helpful actions is to recognize which actions are more likely to be useful at the current state and delay (or completely ignore) the application of other actions. MA-A* is complete, and does not ignore actions. However, different agents will expand the same node at different stages, and when a solution is reached, some expansions that would have taken place in A* may not be applied in MA-A*. As expanding a state entails evaluating its successors, the positive effect of delayed expansions is especially felt when using heuristics such as LM-cut, which are accurate but expensive to compute.

Symmetry exploitation utilizes the notion of public and private actions. As we noted in Corollary 1, the existence of private actions implies the existence of multiple effect-equivalent permutations of certain action sequences. A* does not recognize or exploit this fact, and MA-A* does. Specifically, imagine that agent φ_i just generated state s using one of its public actions, and s satisfies the preconditions of some action a of agent φ_j. Agent φ_i will eventually send s to agent φ_j, and the latter will eventually apply a to it. Now, imagine that agent φ_i has a private action a' applicable at state s, resulting in the state $s' = a'(s)$. Because a' is private to φ_i, from the fact that a is applicable at s we deduce that a is applicable at s' as well. Hence, A* would apply a at s'. However, in MA-A*, agent φ_j would not apply a at s' because it will not receive s' from agent φ_i. Thus, MA-A* does not explore all possible action sequences. If a' (which generates s') is needed in the solution plan, MA-A* will insert it, but not before a (which is a public action of φ_j). Rather, it will insert it before the next public action of agent φ_i. We note that this type of symmetry does not pertain only to MA systems, but to any factored system having internal operators. Therefore, our intuition is that it could be exploited, even in single-agent problems, by re-factoring them into MA ones. How to do this remains an open question.

Perhaps the greatest practical challenge suggested by the distributed version of MA-A* is that of computing a global heuristic by a distributed system. In some domains, the existence of private information that is not shared leads to serious deterioration in the quality of the heuristic function, greatly increasing the number of nodes expanded. We believe that there are techniques that can be used to alleviate this problem. As a simple example, consider a public action

Table 2: Comparison of centralized A* and MAD-A* running on multiple processors. Running time (in sec.), average initial state h-values and efficiency values are shown.

Problem	#	LM-cut heuristic					Merge&Shrink heuristic					Planning First	
		A*		MAD-A*			A*		MAD-A*				
(# of processors)	agents	time	init-h	time	init-h	eff.	time	init-h	time	init-h	eff.	time	δ
logistics7-1(4)	4	55.3	39	178	35	0.08	0	44	57	36	0	–	N/A
logistics8-1(4)	4	24.1	41	172	37	0.04	0.1	44	49.5	37	0	–	N/A
logistics10-0(4)	5	203	41	–	35	0	81.7	43	–	36	0	–	N/A
Rovers3(2)	2	0	11	0	6	1.00	0	9	0	6.5	1.00	0.3	2
Rovers4(2)	2	0	8	0	6	1.00	0	8	0	6	1.00	0.2	1
Rovers5(2)	2	8.8	16	10.4	10.5	0.42	11.7	20	4	10.5	1.46	9.3	3
Rovers6(2)	2	–	23	716	16.5	∞	–	27	325	17.5	∞	–	N/A
Rovers7(3)	3	6.7	15	6.2	11	0.36	55.9	14	7.2	9	2.59	38.5	3
Rovers8(4)	4	–	21	–	13	N/A	–	15	–	10	N/A	–	N/A
Rovers12(4)	4	12.1	16	36.7	9	0.08	119	16	22.2	8.25	1.34	–	N/A
Rovers14(4)	4	–	18	–	10	N/A	–	17	–	11	N/A	–	N/A
satellites5(3)	3	1.3	14	7	8.3	0.06	7	13	3.8	8	0.61	52.3	2
satellites6(3)	3	3.5	17	2.5	8.3	0.47	23.5	18	9.2	9.3	0.85	457	3
satellites7(4)	4	94.5	19	–	9	0	–	14	343	9.5	∞	–	N/A
satellites8(4)	4	–	21	–	10	N/A	–	15	–	10	N/A	–	N/A
satellites9(4)	5	–	22	–	13.8	N/A	–	18	–	11	N/A	–	N/A
satellites10(4)	5	–	26	–	12	N/A	–	17	–	10	N/A	–	N/A
zenotravel9(3)	3	72.1	18	1108	14	0.02	56.7	19	370	14	0.05	–	N/A
zenotravel10(3)	3	16.1	20	–	17	0	26.7	22	–	17	0	–	N/A
zenotravel12(3)	3	458	18	–	14	0	–	–	–	–	N/A	–	N/A
zenotravel13(3)	3	–	22	–	18	N/A	–	–	–	–	N/A	–	N/A

a_{pub} that can be applied only after a private action a_{priv}. For example, in the rover domain, a *send* message can only be applied after various private actions required to collect data are executed. If the cost of a_{pub} known to other agents would reflect the cost of a_{priv} as well, the heuristic estimates would be more accurate. Another possibility for improving heuristic estimates is using an additive heuristic. In that case, rather than taking the maximum of the agent's own heuristic estimate and the estimate of the sending agent, the two could be added. To maintain admissibility, this would require using something like cost partitioning (Katz and Domshlak 2008). One obvious way of doing this would be to give each agent the full cost of its actions and zero cost for other actions. The problem with this approach is that initially, when the state is generated and the only estimate available is that of the generating agent, this estimate is very inaccurate, since it assigns 0 to all other actions. In fact, the agent will be inclined to prefer actions performed by other agents, as they appear very cheap, and we see especially poor results in domains where different agents can achieve the same goal, as in the Rovers domain, resulting in estimates of 0 for many non-goal states.

Finally, the efficiency of existing parallel search methods decreases as the number of processors increases (Kishimoto, Fukunaga, and Botea 2009; Vrakas, Refanidis, and Vlahavas 2001). We have shown that the efficiency of MA-A*, on the other hand, depends more on the structure of the problem than the number of processors. This could mean that in loosely-coupled systems, even extremely large problems could become solvable by adding some computational power, which would not make much difference in the centralized case. For this intuition to be verified, however, a further study regarding MA-A*'s scaling behavior (with respect to the number of processors) must be conducted.

Conclusion

We presented MA-A*, a simple formulation of A* for MA systems. Its parallel and distributed variants are separated only by the way heuristic estimates are computed. The parallel variant, in which agents have complete knowledge of the system, exhibits super-linear speedup on problems where the agents are loosely coupled. In its distributed setting, MA-A* dominated the best (non-optimal) distributed planner, while achieving optimality. In some cases, MAD-A* outperformed its centralized counterpart, despite having less accurate heuristic estimates. We also presented the MA-FD framework for MA planning. MA-FD is based on the successful FD framework, and provides the necessary building blocks for implementing MA planning algorithms and heuristics. Our hope is that having a *simple* and *effective* MA planning framework will increase interest in MA planning research, as FD has done for centralized planning.

References

Brafman, R. I., and Domshlak, C. 2008. From one to many: Planning for loosely coupled multi-agent systems. In *ICAPS*, 28–35.

Burns, E.; Lemons, S.; Zhou, R.; and Ruml, W. 2009. Best-first heuristic search for multi-core machines. In *IJCAI*, 449–455.

Chandy, K. M., and Lamport, L. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3(1):63–75.

Durfee, E. H. 2001. Distributed problem solving and planning. In *EASSS*, 118–149.

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

Helmert, M., and Röger, G. 2008. How good is almost perfect? In *AAAI*, 944–949.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *ICAPS*, 176–183.

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. Int. Res.* 14(1):253–302.

ICAPS. The international planning competition. `http://www.plg.inf.uc3m.es/ipc2011-deterministic/`.

Jonsson, A., and Rovatsos, M. 2011. Scaling up multiagent planning: A best-response approach. In *ICAPS*.

Katz, M., and Domshlak, C. 2008. Optimal additive composition of abstraction-based admissible heuristics. In *ICAPS*, 174–181.

Kishimoto, A.; Fukunaga, A. S.; and Botea, A. 2009. Scalable, parallel best-first search for optimal sequential planning. In *ICAPS*.

Meisels, A. 2007. *Distributed Search by Constrained Agents: Algorithms, Performance, Communication (Advanced Information and Knowledge Processing)*. Springer.

Méro, L. 1984. A heuristic search algorithm with modifiable estimate. *Artif. Intell.* 23(1):13–27.

Nissim, R.; Brafman, R. I.; and Domshlak, C. 2010. A general, fully distributed multi-agent planning algorithm. In *AAMAS*, 1323–1330.

Szer, D.; Charpillet, F.; and Zilberstein, S. 2005. Maa*: A heuristic search algorithm for solving decentralized pomdps. In *UAI*, 576–590.

ter Mors, A.; Yadati, C.; Witteveen, C.; and Zhang, Y. 2010. Coordination by design and the price of autonomy. *Autonomous Agents and Multi-Agent Systems* 20(3):308–341.

Tu, P. H.; Pontelli, E.; Son, T. C.; and To, S. T. 2009. Applications of parallel processing technologies in heuristic search planning: methodologies and experiments. *Concurrency and Computation: Practice and Experience* 21(15):1928–1960.

Vrakas, D.; Refanidis, I.; and Vlahavas, I. P. 2001. Parallel planning via the distribution of operators. *J. Exp. Theor. Artif. Intell.* 13(3):211–226.

Yokoo, M.; Durfee, E. H.; Ishida, T.; and Kuwabara, K. 1998. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Trans. Knowl. Data Eng.* 10(5):673–685.

Introducing a new lifted Heuristics based on Lifted Relaxed Planning Graphs

Bram Ridder & Maria Fox
King's College
Department of Informatics

Abstract

In previous work we have shown that grounding, while used by most (if not all) modern state-of-the-art planners, is not necessary and sometimes even undersirable. In this paper we extend this work by demonstrating how to extract heuristic estimates which does not require us the ground the entire domain. We do this by constructing a *Lifted Relaxed Planning Graph* and extracting a relaxed plan, much in the same way FF does. We show that, although we do not have to ground the domain, we compare favourably to FF in terms of states visited and plan quality. This, we believe, offers scope for creating new least-commitment planners which do not require domains to be fully grounded.

INTRODUCTION

Modern planners rely on *grounding*, enumerating all possible actions, before search can begin. A key reason for this need to ground is computation of heuristics, for example, the *Relaxed Planning Graph* is used by a lot of planners (Helmert 2006; Richter and Westphal 2009; Hoffmann 2001). This grounding phase, however, severely restricts the scalability of planners when considering larger problems: the planning process can often fail and run out of memory before search or heuristic computation even begin. For example, recent work (Flórez et al. 2011) showed that real world logistic problems cannot be handled by domain-independent planners as they run out of memory during grounding.

Attempts have been made in the past to either reduce or forego grounding altogether, especially in Partial-Order planners like NONLIN (Tate 1977), UCPOP (Penberthy and Weld 1992) and VHPOP (Younes and Simmons 2003). The scalability of the current best approaches benefit from highly effective heuristics but are limited by the size of problems they can ground. Although not as competitive the lifted approaches are not limited by this, so there is great untapped potential there if only we could get good heuristics.

The goal of this paper is to introduce an efficient lifted heuristic which does not require the entire domain to be grounded. We do this by constructing a *Lifted Relaxed Planning Graph*. In order to forgo grounding the entire domain we introduce *equivalence relationships* between objects. This is a key step towards improving non-grounded planners to be competitive with more recent approaches. We will explore this claim by comparing our lifted heuristic against FF(Hoffmann 2001).

The paper is structured as follows: Section 2 introduces the key concepts of the analysis performed by TIM (Fox and Long 1998). This will help us to identify objects with similar properties, e.g. trucks that can traverse the same map and consider these as single entities. We extend this analysis and construct the lifted structures used to construct the *Lifted Relaxed Planning Graph* introduced in Section 3. In Section 4 we present our lifted heuristic and compare how it performs compared to the FF heuristic. Finally we conclude with remarks and future work.

Constructing The Lifted Transition Graph

Our aim is to limit grounding as much as possible and identify objects which can be proven to be *equivalent*. Intuitively objects which are only part of a predicate which can only be made true or false are not very interesting, nor are objects which are only part of predicates which are not affected by any operators in the domain. Objects of interest to us are part of *state invariants*, these objects are part of a set of predicates of which only one can be true at any given time (e.g. a truck can only be at a single location at any given time). Different methods have been developed to find these (Bernardini and Smith 2011; Helmert 2009; Edelkamp and Helmert 1999), in this paper we use TIM (Fox and Long 1998) to infer the state invariants. In this section we will introduce the key concepts of TIM and describe how the lifted data structures are generated which will be used by the lifted reachability analysis in the next section. We use the *Depots* domain from the IPC-3 (Long and Fox 2003) as a rolling example throughout this section and we assume the reader is familiar with this domain. In this domain crates are being transported between different places by trucks, at each location hoists can stack / unstack crates on top of pallets or other crates and load / unload crates from trucks.

TIM Analysis

For reference we briefly introduce the key concepts of TIM we will use throughout this paper, this is not meant to be a complete summary which we cannot give due to space constraints. For a full explanation of the concepts and definitions we refer to the TIM JAIR article (Fox and Long 1998).

Definition 1 — Typed Planning Task
A typed planning task is a tuple $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ where:

- T is a set of types. Every type $t \in T$ has a set of supertypes,written $SuperType(t)$.
- O is a set of objects, each object $o \in O$ is associated with a type $t \in T$, written $Type(o)$.
- P is a set of predicates, where a predicate $p \in P$ is a tuple $\langle name, types \rangle$. A variable v is a pair $\langle t, D_v \rangle$, where $t \in T$ and D_v is the domain which is a set of objects. The set is initialized by: $o \subseteq O \mid Type(o) \in SuperTypes(t) \cup \{t\}$. An atom is a tuple $\langle p, V \rangle$, where $p \in P$ and V is a set of variables. We refer to the ith variable with the notation V_i.
- A is a set of operators, where an operator $a \in A$ is a tuple $\langle name, parameters, precs, effects \rangle$. $parameters$ is a set of variables. $precs$ and $effects$ are atoms, which are pairs $\langle p, v \rangle$, where $p \in P$ and $v \subset variables$.
- s_0 is set of grounded atoms called the initial state.
- s_g is set of grounded atoms called the goal.

Definition 2 — Solution to a Typed Planning Task
A solution to a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ is a sequence of grounded operators $\{o_0, \ldots, o_n\}$, where $o_{i_{precs}} \subseteq s_i \mid i \in 0, \ldots, n$ and s_{i+1} is the result of applying the effects of each grounded operator $\{o_0, \ldots, o_i\}$ to s_0 in sequence and $s_g \subseteq s_{n+1}$.

TIM takes a typed problem domain and infers state invariants by constructing a transition rule for each parameter of each operator.

Definition 3 — Property
A property is a predicate subscripted by a number between 1 and the arity of that predicate. Every predicate of arity n defines n properties.

For example, given the predicate $(at\ truck\ place)$, the property at_0 refers to the first term $truck$.

Definition 4 — Transition Rule
A transition rule for operator $o \in A$ and variable $v \in o_{parameters}$ is an expression of the form: $property \Rightarrow property \rightarrow property$ in which the three components are bags of zero or more properties called enablers, start and finish, respectively. The start bag will contain properties of the preconditions which are deleted with respect to v while the finish bag will contain properties of the effects which are added with respect to v.

We are only interested in the start and finish bags and drop the enablers bag in all examples. Transition rules with an empty start or finish bag are called *attribute transition rules*, others are said to *exchange* properties. Due to the way TIM constructs these transition rules, only a single property is exchanged per transition rule.

Example 01 *A transition rule for the operator Drop and variable crate is* $\{lifting_1\} \rightarrow \{at_0, on_0, clear_0\}$. *Meaning that a crate that is lifted can exchange this property by being dropped for being i) At a place; ii) On a surface; and iii) Clear.*

TIM combines these transition rules into *Property Spaces* and *Attribute Spaces*, transition rules are combined if their start or finish bags overlap. Spaces which contain an *attribute transition rule* are Attribute Spaces, while those which do not are called *Property Spaces*. Property spaces define mutex relationships on the properties which can be true for a given object at any time, e.g. the Property Space which includes the transition rule in Example 01 defines that a crate object can either be: i) At a place, on a surface and Clear; ii) Being lifted by a hoist; iii) In a truck; or iv) At a place and have a crate on top.

For the remainder of the paper we refer to properties which are part of a *Property Spaces* as *balanced* and those which are not as *unbalanced*.

Lifted Transition Graph

Given the property spaces and attribute spaces and their associated transition rules found by TIM, we will transform them into *Lifted Transitions*.

Definition 5 — Lifted Transition
Given a transition rule a *Lifted Transition* $\langle op, from, to, free \rangle$ is constructed where:

- $op \in A$ is the operator of the transition rule.
- $from \subseteq op_{precs}$ is a node which contains a subset of the preconditions of op.
- $to \subseteq op_{precs} \bigcup op_{effects}$ is a node which contains a subset of the persistent preconditions and the effects which correspond to the finish bag of the transition rule.
- $free : p \in op_{precs} \mid p \notin from$ is a set which contains all preconditions which are not part of $from$ (A precondition can only be a member of one of these sets).

Lifted transitions are very similar to operators in *Domain Transition Graphs* (DTGs)(Helmert 2006), each lifted transition records how an operator changes the value of a given variable.

Initially the preconditions which are part of $from$ are those which correspond to the properties in the start bag of the transition rule. All other preconditions are initially put into the *free* set.

Example 02 *Given the transition rule:* $\{lifting_1\} \rightarrow \{at_0, on_0, clear_0\}$ *the following* Lifted Transition *is initially constructed:*

- $op = Drop$.
- $from = \{(lifting\ hoist\ crate)\}$.
- $to = \{(at\ crate\ place), (on\ crate\ surf), (clear\ crate)\}$.
- $free = \{(at\ hoist\ place), (at\ surf\ place), (clear\ surf)\}$.

Lifted transition differ from transitions in DTGs in that, for each fact $\langle t, D_v \rangle \in op_{parameters}$, the number of values of each variable D_v is not limited to one (i.e. it is not grounded). To check if the operator op is applicable given a set of facts, we need to: 1) Map every fact to all the preconditions in $from$ and $free$; 2) Find any pair of sets of facts

which satisfy all the preconditions in *from* and *free* respectively; and finally 3) Check if the sum of the pair of sets satisfies all preconditions of op.

If all facts are grounded case 2 and 3 do not need to be considered, but in our case these operations are quite expensive because we need to compute the Cartesian product of the *from* and *free* sets and check which pairs of products satisfy all the preconditions of op. To reduce this overhead we split the preconditions into two sets such that the unification of any set of facts that satisfy the preconditions in *from* with any set of facts that satisfy the preconditions in to satisfies all the preconditions of op. Any set of facts which satisfy a set of preconditions is called consistent. We will now describe how *from* and *free* are constructed such that the union of any pair of consistent sets is consistent with the preconditions of the operator op.

Lifted Transitions for Property Spaces Given a lifted transition $\langle op, from, to, free \rangle$, there are two naive ways to guarantee that the union of any pair of consistent sets $\langle C_{from}, C_{free} \rangle$ for *from* and *free*, respectively, is consistent with op_{precs} are by grounding all variables or by adding all preconditions to *from*. In the former case we reduce the number of values every variable can take to one so no inconsistency can arrise. In the latter the union of a consistent set with an empty set is still consistent. However, we can do much better.

The following rules are enforced:

- For every precondition $\langle p, V \rangle \in free$ for which the property $p_i \mid i \in \{0, \ldots, \mid V \mid\}$ is balanced and there exists a precondition $\langle p', V' \rangle \in from$ for which the property $p'_j \mid j \in \{0, \ldots, \mid V' \mid\}$ is balanced and $V'_j = V_i$, then we we move $\langle p, V \rangle$ to the set *from*.
- If the above holds, except that p_i and p'_j are not balanced we ground that variable.
- Every precondition $\langle p, V \rangle \in op_{precs}$ that is not deleted by an effect we check if there exists a fact $\langle e, V' \rangle \in to$ such that $p_i \mid i \in \{0, \ldots, \mid V \mid\}$ and $e_j \mid j \in \{0, \ldots, \mid V' \mid\}$ are both balanced and $V_i = V'_j$. If this is the case $\langle p, V \rangle$ is persistent and is added to the set to.

The first rule ensures that preconditions that contain a state invariant that is shared with any other precondition in the set *from* are added to the same set. The second rule grounds any variable which is shared between any two facts in *from* and *free* but is not a state invariant. The last rule adds all the preconditions which are not removed and share a state invariant with a fact to the set to.

By constructing the lifted transitions this way consistency is guaranteed. This is because facts which share a state invariant are in the same set and secondly all the variables which occur in both sets are grounded.

When we apply these rules to Example 02, we move $(at\ hoist\ place)$ to the *from* set because the property at_0 is balanced and the variable $hoist$ is shared. This does not apply to $(at\ surface\ place)$ because at_1 is unbalanced, thus we ground the variable $place$. The last precondition $(clear\ surface)$ shares no variables with any of the elements in *from*. Finally, the precondition $(at\ surface\ place)$ is added to to, because the variable $surface$ is shared and at_0 is balanced.

Lifted Transitions for Attribute Spaces At this point we have only defined Lifted Transitions for property spaces. We create the same structures for attribute spaces but the way in which they are constructed is different. We are only interested in attribute spaces which add a fact, this is because we perform relaxed reachability analysis which means no fact is ever deleted, so those attribute spaces which remove atoms are not of interest to us.

Unlike property spaces, no property is exchanged for another so it is unclear how to split the set of preconditions and which variables should be grounded. For every transition rule from an attribute space which has not been processed yet for a property space — it is possible that an operator is part of multiple transitions rules — we create a Lifted Transition $\langle op, from, to, free \rangle$ where:

- op is the operator of the transition rule.
- *from* contains all the preconditions.
- to contains the effect achieved by the transition rule.
- *free* is empty.

None of the variables are grounded. Since all the preconditions are part of the same set consistency is guaranteed.

Merging After creating all the lifted transitions we try to limit the number of *from* and to sets by merging sets which are equivalent. This way we create a graph where the *from* and to sets become the nodes and the lifted transitions which become the edges. The created graphs are reminiscent of the DTGs created by Fast-Downward. An important difference is that a DTG is limited to the exchanged properties of a single object, while we create graphs based on *types* and allow graphs of different types to be connected.

Definition 6 — Node equivalence
Two nodes are equivalent if there exists a bijection of both sets of atoms where the predicate and variable domains of each mapped pair are the same. In addition, two variables of any pair of atoms are equal in one node if and only if the variables corresponding to the mapped atoms are also equal.

The latter requirement of Defition 6 is necessary for some domains where — based on the predicate and variable domains alone — multiple bijections are possible. One such example is the Blocksworld domain: some nodes contain the pair $(on\ block\ block) \wedge (on\ block\ block)$ so the relationships between the variables become important to create the correct bijection. For any set of nodes which are equivalent, we select one node from this set n and map all the lifted transitions from the other nodes in the set to n.

In addition we also look for nodes which are equivalent except one node contains facts which are lifted while the mapped fact is grounded. In these cases we ground the lifted variables which are mapped to grounded variables so that these nodes can be merged as well.

Finally we look for nodes which are equivalent except that one contains facts whose predicates contains supertypes of the mapped fact's predicates. If the sets of operators of both

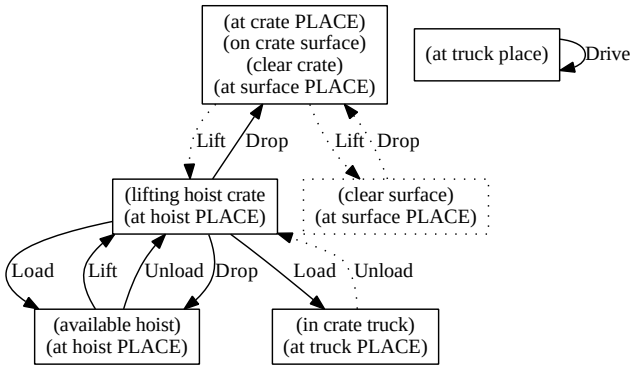


Figure 1: Depots LTG - dotted nodes and edges are removed

nodes are equivalent we merge the node containing the subtypes with the nodes containing the predicates with supertypes. This further reduces the number of lifted transitions and nodes in the graph. For example, the Depots domain contains the types $crate$ and $surface$, $crate$ being a subtype of $surface$. The operator $drop$ can be applied to both $types$ and the lifted transitions for both types are identical. As such we remove the lifted transition with the operator $drop$, which applies only to the type $crate$, in favour of the more general lifted transition which is applicable to the type $surface$.

Figure 1 shows an example for the Depots domain. The terms which are fully capitalised have been grounded.

Remove equivalent transitions The last step in constructing the *Lifted Transition Graph* is removing all lifted transitions which are equivalent. For any pair of lifted transitions with the same operator op whose union of $from$ and $free$ are identical, one can be removed. Given the set of identical *Lifted Transitions* we try to remove the transitions in such a way that we maximise the number of nodes which are not part of any transition. These nodes can be removed.

For example, for the Depots domain we construct the following lifted transition where fully capitalised terms have been grounded:

- $op = Unload$.
- $from = \{(in\ crate\ truck), (at\ truck\ PLACE)\}$.
- $to = \{(lifting\ hoist\ crate), (at\ hoist\ PLACE)\}$.
- $free = \{(available\ hoist), (at\ hoist\ PLACE)\}$.

However, in Figure 1 we can see that a lifted transition with $unload$ as an operator already exists with $from = \{(in\ crate\ truck), (at\ hoist\ PLACE)\}$. In fact, that lifted transition is identical to the one above when we swap the $from$ and $free$ sets thus the above lifted transition is removed as depicted in Figure 1.

Comparing the number of lifted transitions to all the grounded transitions we notice that in the worst case we define exactly as many transitions, but in most cases the number of lifted transitions is a multitude of magnitudes smaller, as can be seen in Table 1.

	Zeno	Satellite	Storage	Blocksworld
Lifted Case	61	5	348	6
Grounded Case	959530	43290	348660	612
	Driverlog	Gripper	Depots	Rovers
Lifted Case	529	7	57	940
Grounded Case	218300	6204	55936	423064

Table 1: Comparison of the total number of transitions

Lifted Relaxed Planning Graph

We now introduce the *Lifted Relaxed Planning Graph* (LRPG) and discuss how we construct it. This graph is used to extract the heuristic estimates as described in the next section. The LRPG is very similar to the (grounded) *Relaxed Planning Graph* (RPG) in fact, when performing reachability analysis on both graphs by constructing them to the level-off point and treating all the facts in the last fact layer as reachable, both graphs will produce the same output (Ridder and Fox 2012).

First we introduce equivalence relations between objects and how these are established, this enables us to reason about lifted facts. Next we discuss how the LRPG is constructed.

Equivalent Objects

In order to perform lifting, we are looking for equivalent objects. Such a relationship exists between objects when they can be used interchangeably without breaking the soundness and completeness of the reachability algorithm. Given two equivalent objects, $o_1 \in O$ and $o_2 \in O$, and an atom, $\langle p, V \rangle$ then the atom $\langle p, V' \rangle$ is also reachable, where

$$V'_j = \begin{cases} \{o_1, o_2\}, & \text{if } o_1 \in V_j \vee o_2 \in V_j \\ V_j, & \text{otherwise} \end{cases} \quad (1)$$

To demonstrate how these relationships are established we introduce an *Annotated Object* structure for every object and combine those which are equivalent in an *Equivalent Object Set*.

Definition 7 — Annotated Object
An *Annotated Object* is a tuple $\langle I, o, G \rangle$, where:

- I contains each initial fact $\langle p, V \rangle \in s_0$, for which $\exists_{v \in V} o \in v$.
- $o \in O$ is the object this *Annotated Object* is created for.
- G is the *Equivalent Object Set* this *Annotated Object* is part of and contains all the *Annotated Objects* it is equivalent with.

Definition 8 — Equivalent Object Set
An *Equivalent Object Set* is a tuple $\langle E, R, f \rangle$, where:

- E is a set of *Annotated Objects* that are *equivalent*.
- R is the set of *lifted facts* that are reachable and contain this *Equivalent Object Set*.
- f is a *signature*. Given the type of the object of the *Annotated Object* this *Equivalent Object Set* was created for,

a signature is created, unique to the set of variables of all operators whose type is equal to, or a supertype of, the type of the object. *Equivalent Object Sets* can be made equivalent *iff* their signatures match.

Definition 9 — Lifted Fact
A lifted fact is a tuple $\langle p, E \rangle$, where:

- $p \in P$ is a predicate.
- E is a set of *Equivalent Object Sets* where the size of the set is equal to the arity of the predicate p.

We can conclude that two *Equivalent Object Sets* $eos1$ and $eos2$ are equivalent if there exist two *Annotated Objects* $ao1 \in eos1$ and $ao2 \in eos2$ such that ${eos1}_R$ is a superset of ${ao2}_I$ and ${eos2}_R$ is a superset of ${ao1}_I$. In other words, if each can reach the initial state of the other we conclude that the *Equivalent Object Sets* they are part of must be equivalent. Thus by proving that two members of both sets are equivalent we have proven that this relation holds for all members of both sets.

Establishing these relationships will create an overhead during the construction of the LRPG, but once these relationships have been established it can speed up finding a relaxed plan significantly.

A good illustration of this can be found in the *Depots* domain. Truck objects have no other relation to other objects, except for the place they are at: all the truck objects at the same location are equivalent. This is closely related to functional symmetry (Fox and Long 1999) and other symmetry relationships defined in the literature (Rintanen 2003). However, we go much further than this. The *Drive* operator does not restrict where trucks can drive to, so we apply this operator once to all truck objects to make the fact $(at\ truck\ place)$ true for all truck and place objects. Assuming no crate object was in any truck in the initial state, all truck objects are now equivalent. Crate objects require more steps to make them equivalent because the initial state typically defines a surface they are placed on and another crate that is stacked on top of them.

Creating the LRPG

We will now explain how we construct an LRPG. The pseudo code is shown in Algorithm 1. We shall now discuss every step in greater detail.

Initialise The starting point of constructing the lifted relaxed planning graph is the atoms in the initial state s_0. These are converted into lifted facts and added to the current fact layer. Initially, every equivalent object set contains a single Annotated Object and the set is constructed from the set of objects O.

Update Equivalent Objects The next step in the algorithm is to establish all possible equivalence relationships between objects, this is done by the *updateEquivalences* function. For any two objects which can be made equivalent we merge their corresponding *annotated objects* into a single equivalent object set. We keep a backlog of which objects are proven to be equivalent per fact layer. We will explain why this is necessary when we explain how to extract a relaxed plan from the LRPG. Finally we update all the lifted facts with respect to the updated equivalent object sets and remove all duplicate facts from the current fact layer.

Algorithm 1: Constructing a Lifted Relaxed Planning Graph till Level-Off Point

```
for o ∈ O do
    Create a new Equivalent Object Set for o;
c_action_layer ← null;
c_fact_layer ← FactLayer();
for i ∈ s_0 do
    c_fact_layer → add(i);
updateEquivalences(c_fact_layer);
done ← False;
while ¬done do
    c_action_layer ← ActionLayer(c_fact_layer);
    n_fact_layer ← FactLayer(c_action_layer);
    done ← True;
    for l ∈ Lifted Transitions do
        if generateReachableFacts(l) then
            done ← False;
    c_fact_layer ← n_fact_layer;
    updateEquivalences(c_fact_layer)
```

Procedure generateReachableFacts(l)

```
consistent_sets ← createSets(l, c_fact_layer);
for C ∈ consistent_sets do
    operator ← substitute(l.operator, C);
    c_action_layer add(operator);
    for e ∈ operator → effects do
        n_fact_layer →add(e);
return True if a new fact was generated, False otherwise
```

Generate Reachable Facts Given all the facts F which are true in the current fact layer (c_fact_layer) and the set of lifted transitions T in the lifted transition graph, then we map every fact $f \in F$ to each precondition $p \in \bigcup_{t \in T} from(t) \cup free(t)$ it can unify with. Next we take the Cartesian products of all mapped facts for each set $from(t)$ and $free(t)$ and store the consistent mappings in $M_{t_{from}}$ and $M_{t_{free}}$, respectively. The atoms which are made true by executing the operator t_{op} given the preconditions $(m_{from} \in M_{t_{from}}) \cup (m_{free} \in M_{t_{free}})$ are added to the next fact layer (n_fact_layer). In addition we also copy all the lifted facts from the current layer to the next layer by executing a special operator *NOOP*, this operator has the copied lifted fact as both its precondition and effect.

Note that the pseudo code — while producing the same output — is not indicative of how this function is actually implemented, it is presented this way for clarity.

While it is true that the facts in the intitial state are grounded, we should not expect that all the facts in every

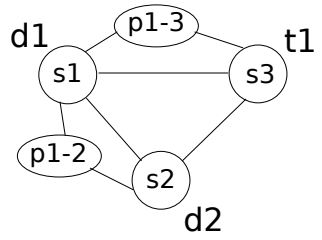


Figure 2: Driverlog example domain

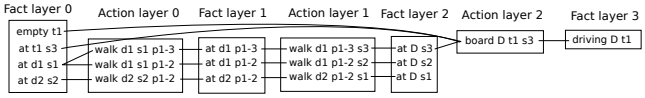


Figure 3: Driverlog LRPG; $D = \{d1, d2\}$

fact layer are therefore grounded too.

The reason for this is twofold; Firstly we establish equivalence relationships between objects and secondly not every effect is constrained by the preconditions of an action. Consider, for example, the *to* parameter of the *Drive* operator from the Depots domain. It is not restricted by any precondition which means any truck can drive to any place.

Example 03 *Drive - $truck, from, to$*
Preconditions: $(at\ truck\ from)$
Effects: $\neg(at\ truck\ from) \wedge (at\ truck\ to)$

In these situations, given an unrestricted parameter $\langle t, D_v \rangle \in a_{parameter} \mid a \in A$ and a lifted fact l mapped to any of the preconditions $\langle p, V \rangle \in a_{precs} \mid \exists_{i \in \{0,\ldots,|V|\}} V_i = \langle t, D_v \rangle$, then we add to the set l_{E_i} each annotated object ao where $Type(ao_o) = t \vee t \in SuperType(ao_o)$.

For every generated effect e we check if it has been generated before, if there exists a lifted fact which can be unified with e, we add the *operator* to its set of achievers. Otherwise, we add it to the next fact layer. In addition, e will be added to every Equivalent Object Set $eog \in e_E$, this is done to prove future equivalence relationships.

Example 04 *As an example of how a LRPG looks like after construction, consider the Driverlog domain depicted in Figure 2. We have two drivers who can walk over the path $s2 \leftrightarrow p1-2 \leftrightarrow s1 \leftrightarrow p1-3 \leftrightarrow s3$ and board the only truck in the problem located at location $s3$. The goal of the domain is to achieve the fact $(driving\ d2\ t1)$, the constructed LRPG is depicted in Figure 3. Please note that we have removed all the static facts, like the connection between the paths, and only list the first achiever of any fact for readability.*

Till fact layer 2 the LRPG is identical to that of the grounded RPG. However, when reaching fact layer 2 we can make both drivers equivalent because both drivers have reached the initial location of the other. Thus in fact layer 2 both drivers can be used interchangeably which means that we now know that driver $d2$ can reach the fact $(at\ d2\ s3)$ even though no action in action layer 1 achieves this fact. In general this means that the number of fact layers in the LRPG will at most be equal to that of a grounded RPG but more often than not contains considerable fewer. As detailed in previous work(Ridder and Fox 2012) performing reachability analysis based on these LRPGs consumes far less resources and is quicker than the grounded approach too, making it a more viable tool for larger domains because it *scales* better as the problem instances become larger.

Calculating the Heuristic

The question that has, until now, remained open is: how good are the heuristic estimates we can derive from the LRPG? To test this we will perform a direct comparison to the FF heuristic and mimic it as closely as possible but there are some key differences which we will discuss in this section. The reader is assumed to have a working knowledge of the FF heuristic (Hoffmann 2001).

Given a planning problem Π we begin, like FF, by constructing a LRPG until we find a fact layer which contains all goal atoms Π_{s_g}. From this graph we construct a relaxed plan by adding all facts we want to achieve in a priority queue and then picking an achiever for every fact in this queue. The facts to achieve are ordered by beginning with the facts which appear latest in the LRPG. When multiple achievers can achieve a given fact we use the same preferences as FF; 1) Always prefer a NOOP when one is available; 2) If a NOOP is not available we give each achiever an estimate of how difficult it is to achieve its preconditions. This estimate is calculated by summing the fact numbers for each of the achievers preconditions, where the layer number is taken from the first layer where the precondition first appears.

The above description is how FF extracts a relaxed plan from an RPG. Unfortunately, if we were to apply the above algorithm and derive a heuristic from the LRPG, it would not be very informative. To explain why this is the case we refer back to the LRPG constructed for the driverlog domain depicted in Figure 3. Following the rules above yields following relaxed plan:

- *(walk d1 s1 p1-3)*
- *(walk d1 p1-3 s3)*
- *(board {d1,d2} t1 s3)*

The astute reader will have realised that the relaxed plan above solves the problem of achieving $(driving\ d1\ t1)$ (when we bind $\{d1, d2\}$ to $d2$). The reason why this happens is because we prove that $d1$ and $d2$ are equivalent before any of these drivers get into the truck. Thus, when extracting the relaxed plan from the LRPG it will trace back to the driver which gets there first. This might not be the driver we want. To solve this issue and get better heuristic guidance, we need to do two things.

First of all we need to *bind* the variable domains of the chosen achiever so it respects the bindings of the fact it achieves. In our example when we pick the board operator as an achiever we need to bind $\{d1, d2\}$ to $d2$. While in this example this equates to grounding the achiever, this is normally not the case. For example if the goal was to achieve *(at t1 s1)* and the cheapest achiever was *(drive {d1, d2} t3*

s1) , there would be no need to bind the variable referring to which driver should drive the truck.

When we apply the necessary binding the relaxed plan looks like this:

- *(walk d1 s1 p1-3)*
- *(walk d1 p1-3 s3)*
- *(board d2 t1 s3)*

Here we notice that something strange is going on: we enforce that driver $d2$ should get into the truck, but the preconditions for this achiever are linked to a different driver, $d1$. We need to substitute $d2$ for $d1$ in order to make the relaxed plan valid. Whenever we need to make a substitution between different equivalent object sets, we effectively lose information because we are solving a different relaxed problem.

To retrieve part of this information we need to augment the heuristic value by adding the cost of making two object sets equivalent. In this case we want to add the cost of making $d2$ equivalent to $d1$. One way of doing this is by solving the following planning problem:

Definition 10 — Equivalence Planning Problem Given a planning problem $\Pi = \langle T, O, P, A, s_0, s_g \rangle$ and two equivalent objects o_1 and o_2, then we define F as the set of all initial atoms which contains either object $F : \{\langle p, V \rangle \in s_0 \mid \exists_{v \in V} o_1 \in D_v \vee o_2 \in D_v\}$. We then change the variable domains $v \in V$ of every atom $\langle p, V \rangle \in F$ such that:

$$v = \begin{cases} o_1, & \text{if } D_v = \{o_2\} \\ o_2, & \text{if } D_v = \{o_1\} \\ v, & \text{otherwise} \end{cases} \quad (2)$$

The *equivalence planning problem* is then defined as $\Pi' = \langle T, O, P, A, s_0, F \rangle$.

While this would work, it might be quite costly to use this method since, because on a reasonable sized problem we might need to solve many of these additional *equivalence planning problems*. Also, while solving an equivalence planning problem, we might run into the same issue. Therefore, as a simple approximation, we take the number of the fact layer at which the equivalent object sets were made equivalent as a supplement to the heuristic estimate, and add it to the length of the extracted relaxed plan.

In our example the equivalent object sets of both drivers become equivalent at fact layer 2, the total heuristic therefore becomes 5 which, incidentally, is the length of the optimal plan. While this works well in this particular example, there are examples where our heuristic performs less well than the FF heuristic. For example the problem in Figure 4 is to get $d1$ to location G. The length of each dotted path is n. The problem with this structure is that $d2$ will be made equivalent with $d1$ at fact layer $2n+1$, but *(at d2 G)* is made true at fact layer $2n$. Thus, when extracting a relaxed plan we find a plan of length $2n$, but then we need to substitute $d1$ for $d2$ which will cost $2n+1$ for a total heuristic value of $4n + 1$. By contrast, FF will find the correct heuristic value of $2n + 1$.

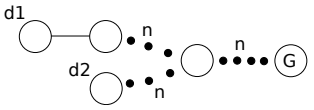


Figure 4: A case where our heuristic performs poorly

Results

We will now present results showing how well our lifted heuristic estimate works by comparison to FF. We used the same codebase as FF and modified it so that it uses our heuristic function. We compared the number of states explored under the two heuristics, as well as the quality of the plans found.

We run all our experiments on an Intel Core i7-2600 running at 2.4GHz and allowed 4GB of RAM, we allowed for 10 minutes of computation time. We have taken 7 problem domains from various planning competitions, and we now discuss the results obtained in these domains.

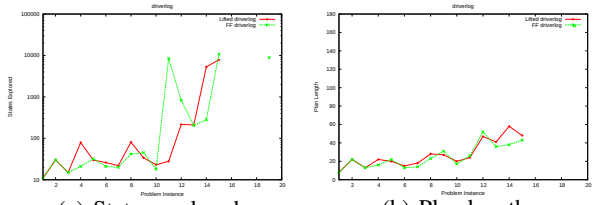
Driverlog

The results for the Driverlog domain are depicted in Figure 5(a) and Figure 5(b). The results are a mixed bag, for some problem instances we produce better quality plans and faster but for others we do not. The reason for this is twofold. First of all, recall the example we gave in the Section *Calculating the Heuristic* if the problem links the various locations is such a way it will misdirect the search effort. Refering back to Figure 4, if our goals is to get $d1$ to G we misguide the heuristic because moving $d2$ towards the goal discreasese the heuristic value by two whereas moving $d1$ will only decrease the heuristic value by one. This is why the number of states explored is higher whenever more states are explored by our heuristic.

On the other hand, the relationship between the packages and the trucks in the domain can help us get a better heuristic estimate. Imagine a problem domain where we have four locations, s1, s2, s3 and s4 which are all connected as follows: $s1 \leftrightarrow s2 \leftrightarrow s3 \leftrightarrow s4$. In the initial state a truck is at $s3$, a package $p1$ starts at the same location and needs to be delivered to $s4$ while a second package $p2$ starts at $s1$ and needs to be delivered at the same location. The relaxed plan generated by FF will load $p1$, drive the truck in until it reaches all locations, load $p2$, and unload both packages at $s4$ — a heuristic value of 7. Our heuristic will produce the same results, with the exception that both packages will be made equivalent and we need to substitute $p1$ for $p2$ which adds 4 to the heuristic totalling 10, which is closer to the optimal heuristic of 9.

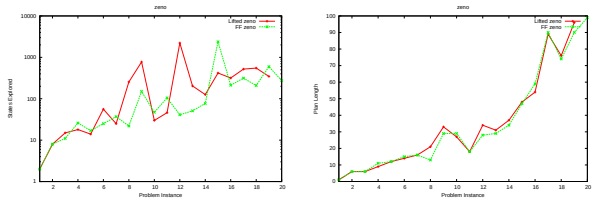
Zeno

The results for the Zeno domain are depicted in Figure 6(a) and Figure 6(a). The Zeno domain does not contain any constraints where planes can fly to except for their fuel level. It is therefor not surprising that the states explored and plan quality do not differ significantly from that of the FF heuristic.



(a) States explored (b) Plan length

Figure 5: Driverlog

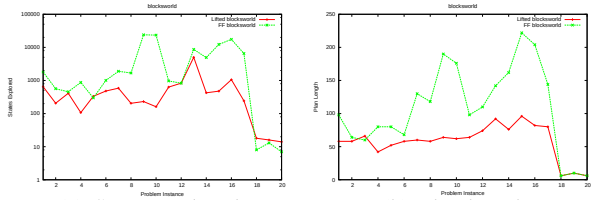


(a) States explored (b) Plan length

Figure 6: Zeno

Blocksworld

The results for the Blocksworld domain are depicted in Figure 7(a) and Figure 7(b). On this domain our heuristic outperforms FF in terms of plan quality and number of states visited. This is because the added costs for the substitution proves to be a very good heuristic estimate. For example, consider the following problem where we have blocks A, B, and C stacked in that order and we want to achieve the goal where A is on top of C and C is on top of B. The lifted heuristic will unstack C from B, B from A and finally pick up A from the table, to achieve the final goal we have to substitute A for C and C for A which gives us the heuristic estimate of 9. Compare this to the FF heuristic which is 4 and the optimal heuristic which is 8.

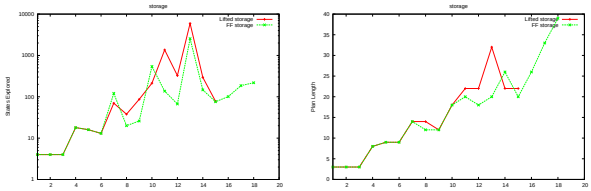


(a) States explored (b) Plan length

Figure 7: Blocksworld

Storage

The results for the Storage domain are depicted in Figure 8(a) and Figure 8(a). This domain contains a lot of static facts which differ for many of the objects defined in the domain. This means that very few objects can be shown to be equivalent, this explains why for quite a few domains we see the exact same results between both heuristic approaches. After all, if we can prove no object equivalences our heuristic approach is similar to FF. In other cases we notice that we often substitute crates before they land on their final destination, much in the way as in the Blocksworld domain. In some cases (like in problem file 07) this gives us a benefit as substituting crates augments the heuristic in such a way that it makes up for the fact that a hoist has to move back and forth to deliver the crates to their final destination. Unfortunatelly this does not always pan out well, which explains why we cannot solve some domains which FF can.

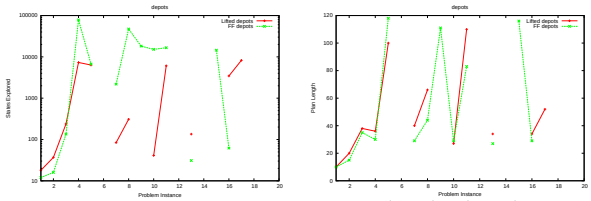


(a) States explored (b) Plan length

Figure 8: Storage

Depots

The results for the Depots domain are depicted in Figure 9(a) and Figure 9(b). On quite a few domains our lifted approach needs to visit considerably fewer states, but the plan quality does suffer on most domains. The reason for this is that, much like the Blocksworld domain, crates can be made equivalent and compensate for some of the delete effects which are ignored. However, unlike the Blocksworld domain where the table is an infinite surface, the surfaces we can stack crates on are finite and distributed over many locations. In some cases our lifted heuristic misdirects the search by bringing crates closer to its initial states but moving away from solving the actual problem.



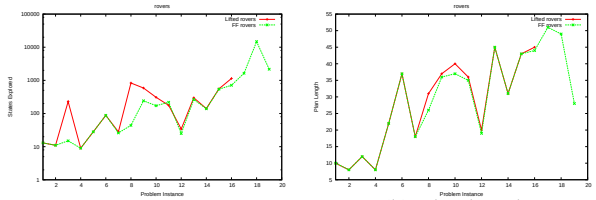
(a) States explored (b) Plan length

Figure 9: Depots

Rovers

The results for the Rovers domain are depicted in Figure 10(a) and Figure 10(b). Like the Storage domain we cannot prove many equivalent relationships for most of the problem instances in this domain. This is due to the static facts which dictates which equipment different rovers have on board and what waypoints they can traverse. It is therefor not surprising that we generate the same plans as FF on most problem instances. The number of states visited does differ, when examinating the results we found that our lifted heuristic estimates were generally lower. This is because we prefer to reuse actions in the relaxed plan, whereas FF selects action based on first layers its preconditions appear in. This gives us an incentive to reuse the same rover for different jobs leading to longer plans and more states to explore in some

cases. For the larger domains we found that we ran out of time before we could explore as many states as FF does, this is because we use a different internal encoding of state spaces and facts than FF does which means that for every heuristic computation we spend time considerable converting these internal representations.

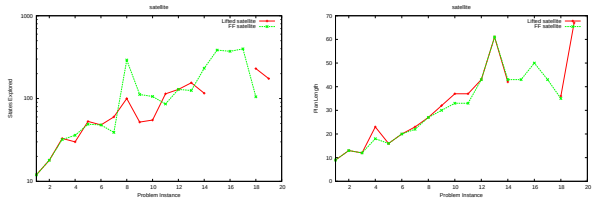


(a) States explored (b) Plan length

Figure 10: Rovers

Satellite

The results for the Satellite domain are depicted in Figure 11(a) and Figure 11(b). Like the rovers in the Rovers domain the satellites can not be made equivalent in most problem instances due to different instrument configuration, but unlike rovers our plans are vastly different from those generated by FF. The reason for this is that the relaxed plans extracted by FF — when having identical satellites to its deposal like on problem file 04 — tend to use one of them to reach the goal while the relaxed plans generated by our lifted heuristic tend to utilise both of them. This is because when we look for an achiever for fact like *(have_image Star5 image1* we will select the achiever *(take_image satellite1, satellite2, ..., satelliten Star5 instrument1, instrument2, ..., instrumentn image1)* and we only have to consider which satellite and which instrument at a later state. We can see from the graph that in quite a few cases this gives us a better heuristic estimate, but at the same time also produces better plans. The reason why our plan length is higher on some problems is because we utilise more satellites compared to FF, leading to more actions but most of them can be executed in *parallel*.



(a) States explored (b) Plan length

Figure 11: Satellite

Conclusions

In this paper we have demonstrated a new lifted heuristic and compared it to its grounded equivalent in the FF framework. We have discussed the weaknesses of this heuristic and shown when the heuristic estimate outperforms its FF equivalent. In previous work we have shown that performing reachability analysis on the lifted structure uses less memory and is quicker on most domains tested. With this preliminary work we have shown that this lifted approach is able to produce informative heuristics and compares favourably to FF. We hope further research in this direction will yield planners which are better scalable and solve larger problem instances than currently can be tackled due to grounding.

Future work in this direction will explore different ways of computing heuristics, including utilising the *Lifted Transition Graph* to calculate heuristics much in the same way as Fast-Downward does. We have good hopes for this approach, because while solving $SAS^+ - 1$ tasks is **NP**-complete(Helmert 2004), some classes can be solved in polynomial time. For example, if the number of low-level variables is bound to a constant. While most planning domains do not have this property we might be able to exploit the equivalence relationships between objects, reducing the number of low-level variables we need to consider.

References

Bernardini, S., and Smith, D. 2011. Automatic synthesis of temporal invariants.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *In ECP*, 135–147. Springer.

Flórez, J. E.; de Reyna, Á. T. A.; García, J.; López, C. L.; Olaya, A. G.; and Borrajo, D. 2011. Planning multi-modal transportation problems. In *ICAPS*. AAAI.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367–421.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proceedings of IJCAI'99*, 956–961.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. *Proceedings of the Fouteenth International Conference on Automated Planning an Scheduling ICAPS 2004* 161–170.

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. 2001. FF: The fast-forward planning system. *AI magazine* 22:57–62.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research* 20:1–59.

Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A sound, complete, partial order planner for ADL.

Richter, S., and Westphal, M. 2009. The LAMA planner. In *ICAPS-06*.

Ridder, B., and Fox, M. 2012. Performing a lifted reachability analysis for improved scalability. In *Technical Report, King's College London*.

Rintanen, J. 2003. Symmetry reduction for sat representations of transition systems. In *ICAPS*, 32–41. AAAI.

Tate, A. 1977. Project planning using a hierarchical non-linear planner. Technical Report 25, Dept. of Artificial Intelligence, Edinburgh University.

Younes, H. L. S., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research* 20:405–430.

Symbolic A* Search with Pattern Databases and the Merge-and-Shrink Abstraction

Stefan Edelkamp and Peter Kissmann
TZI Universität Bremen, Germany
{edelkamp, kissmann}@tzi.de

Álvaro Torralba
Planning and Learning Group
Universidad Carlos III de Madrid, Spain
atorralb@inf.uc3m.es

Abstract

The efficiency of heuristic search planning crucially depends on the quality of the search heuristic, while succinct representations of state sets in decision diagrams can save large amounts of memory in the exploration. BDDA* – a symbolic version of A* search – combines the two approaches into one algorithm. This paper compares two of the leading heuristics for sequential-optimal planning: the merge-and-shrink and the pattern databases heuristic both of which can be compiled into a vector of BDDs and be used in BDDA*. The impact of optimizing the variable ordering is highlighted and experiments on benchmark domains are reported.

Introduction

Explicit-state heuristic search planners have shown advantages to symbolic planners with binary decision diagrams (BDDs) (Bryant 1985) in cost-optimal planning, suggesting that the increased quality of search heuristics sometimes exceeds the structural savings for representing and exploring large state sets in advanced data structures.

For the automated construction of search heuristics for BDD-based planning, *symbolic pattern databases* (PDBs) have been proposed (Edelkamp 2005). They correspond to a complete (or partial) backward exploration of the concrete (or abstracted) state space. These planning heuristics can be exploited in a symbolic version of A* search, BDDA* for short (Edelkamp and Reffel 1998).

Merge-and-shrink (M&S) is among the strongest heuristics for explicit-state space planning (Helmert, Haslum, and Hoffmann 2008). Newer proposals further improve its quality (Nissim, Hoffmann, and Helmert 2011; Katz, Hoffmann, and Helmert 2012) and outperform other state-of-the-art heuristics like *LM-cut* (Helmert and Domshlak 2009) on a sizable number of domains. Furthermore, it can compute perfect heuristics for some simpler benchmark domains in polynomial time.

In this paper we extract the memory structure of the M&S heuristic in form of an algebraic decision diagram (ADD) (Bahar et al. 1997). This allows to enrich a symbolic heuristic planner to exploit this expressive estimate. The precomputed ADD is converted to a vector of BDDs and plugged into BDDA* for computing cost-optimal plans. It exactly matches the estimate quality of the explicit-search variants and is general to all existing M&S variants. We will also look at refinements to BDDA* and propose List BDDA*, which exploits a list representation of the search frontier (rather than a matrix representation).

PDBs perform surprisingly well compared to M&S. In our experiments the former yield the perfect heuristic in more instances than the latter. In several cases the construction does not even need to perform abstraction, but resorts to (possibly truncated) backward search in the concrete state space. While the M&S heuristic is strictly more informed than the PDB heuristic (Helmert, Haslum, and Hoffmann 2008) in case of explicit-state search, this seems not necessarily to be the case in symbolic search. However, we illustrate that there are examples for exponential gaps between the M&S and the PDB heuristics that fail to work for a symbolic construction. Furthermore, we show that ADD reduction can yield smaller structures than the one applied in the M&S abstraction. We will also see that the variable ordering in the two heuristics is a crucial parameter to the exploration and produces outcomes of large variety.

The paper is structured as follows. First, we reconsider explicit-state and symbolic heuristic search. Next, we turn to the set of refinements including a new BDDA* version and to the extraction of the M&S heuristic as an alternative to the symbolic PDB heuristic. Limits and possibilities are discussed. In the experimental results we compare the two heuristics and discuss the outcome and effects of changing the variable ordering in both cases.

Heuristic Search Planning

A *planning task* consists of variables of finite domain so that states are assignments to the variables, an initial state, the goal, and a finite set of operators each being a pair of preconditions and effects. In *cost-based planning*, operators are associated with action cost values, which often are integers – or alternatively rational numbers, which can be cast to integer values. The task is to find a path (the *plan*) from the initial state to the goal. The plan is *optimal*, if its cost is smallest among all possible plans. A *heuristic* is a mapping from states to natural numbers or infinity, and admissible if for all possible states the value is not greater than the cost of an optimal plan. We refer to a finite domain variable encoding of the planning problem, abbreviated as SAS^+ *planning* (Bäckström and Nebel 1995). A planning task *abstraction* is a planning task based on a mapping for the initial state, the

goal state as well as the operators. We consider two heuristics based on abstraction.

Pattern Database

The pattern database (PDB) heuristic, inspired by a selection of tiles in the sliding-tile puzzle (Culberson and Schaeffer 1998), has been extended to the selection of state variables in other domains and in planning (Edelkamp 2001). More general definitions have been applied, shifting the focus from the mere selection of a subset of the SAS$^+$ variables to different state-space *abstractions* that are computed prior to the search. A PDB stores the shortest path distance from each abstract state to the set of abstract goal states. *Partial PDBs* (Anderson, Schaeffer, and Holte 2007) refer to not conducting the backward search to completion but truncating the search at goal distance d and assigning all remaining states the heuristic value $d+1$. As a slightly better estimate, we can take the minimum value of $(i + j) > d$ of the goal distance i of a state within the PDB plus the cost j of an operator. *Implicit PDBs* (Katz and Domshlak 2010) are a refinement imposed by different decompositions of the causal graph.

Merge-and-Shrink

The merge-and-shrink (M&S) heuristic (Helmert, Haslum, and Hoffmann 2008; Nissim, Hoffmann, and Helmert 2011) is induced by a *distance-preserving abstraction*, originally proposed in the context of directed model checking (Dräger, Finkbeiner, and Podelski 2006; 2009). The abstract state space in this heuristic is built incrementally. The rough idea is that SAS$^+$ variables are greedily chosen to construct a larger state space by computing the (synchronous) product of the existing state space and the one induced by the next SAS$^+$ variable.

If the state space becomes too large pairs of states are unified[1]. The approach is layered, so that the union of two state sets is realized by changing the mapping from the state set in one layer to the state set in the next layer. There are different strategies to compute the cross-product state spaces. Most current proposals work on a *linear* arrangement, meaning that one variable is added at a time. Non-linear arrangements that combine arbitrary disjoint variables support sets (limiting the size of the product space) are involved but may yield stronger union operations.

The shrinking is based on the notion of *bisimulation*. Two states s and s' are bisimilar if they agree on whether or not the goal is true and every planning operator leads to the same abstract state from both s and s'. If only bisimilar states are aggregated, then M&S is guaranteed to be perfect. The bisimulation shrinking strategy computes the coarsest bisimulation, and in the shrinking step it aggregates only bisimilar (abstract) states. In most benchmark domains, however, coarsest bisimulations are still large even under operator projection.

[1]The alternative term *merge* conflicts with the name of the heuristic as the step of merging the states is actually referred to as *shrinking*, while the construction of the product state space graph is referred to as *merging*.

Greedy bisimulation (gop) is a relaxed variant of bisimulation, which demands the bisimulation property only for transitions (s, s'), where the abstract goal distance from s' is at most as large as the abstract goal distance from s. This relaxation forfeits the guarantee of providing a perfect estimate.

Motivated by the size of bisimulations, a more approximate shrinking strategy (gop') builds the coarsest bisimulation and keeps unifying states until the size limit M is reached. The latter may happen before a bisimulation is obtained, in which case it looses information. The strategy attempts to make errors only in more distant states, where the errors will hopefully not be as relevant.

Symbolic A* Search

The main limitation for applying PDBs in search practice is the restricted amount of RAM. For the exploration of large state spaces, symbolic search can save huge amounts of memory and computation time. State sets (Pang and Holte 2011) are represented and modified by accessing their characteristic functions.

Decision diagrams (Wegener 2000; Bahar et al. 1997; Bryant 1985) are a memory-efficient data structure used to represent Boolean (or integer-valued) functions as well as to perform set-based search, where the diagram represents all binary state vectors that evaluate to certain values. More precisely, a BDD (an ADD) is a directed acyclic graph with one root and two (several) terminal nodes, called sinks. Each internal node corresponds to a binary variable of the state vector and has two successors (low and high), one representing that the current variable is false and the other representing that it is true. For any assignment of the variables on a path from the root to a sink the represented function will be evaluated to the value labeling the sink. Moreover, decision diagrams are unique for a fixed variable ordering by applying the two reduction rules of (1) eliminating nodes with the same low and high successors and (2) merging two nodes representing the same variable that share the same low successor as well as the same high successor.

In order to perform symbolic search we need two sets of variables, one (x) representing the current states and another (x') representing the successor states. To find the successors of a set of states S represented in the current state variables given a BDD T (the transition relation) for the entire set of operators we use the $image$ operator, i. e., $image(S, x) = \exists x.S(x) \wedge T(x, x')[x' \leftrightarrow x]$, where $[x' \leftrightarrow x]$ denotes the swap of the two sets of variables. Similarly, we can perform search in backward direction by using the $pre\text{-}image$ operator, i. e., $pre\text{-}image(S, x') = \exists x'.S(x') \wedge T(x, x')[x \leftrightarrow x']$.

Symbolic PDBs (Edelkamp 2005) are PDBs that have been constructed symbolically as decision diagrams for later use either in symbolic or explicit heuristic search. Their construction exploits that the transition relation is defined as a relation. The savings observed by the symbolic representation are substantial for many planning domains. Different to the posterior compression (Ball and Holte 2008), the construction in (Edelkamp 2005) works on compressed representation, allowing much larger databases to be constructed. For such *PDB construction*, backward symbolic

search is used. In the case of partial PDBs, the construction is truncated at some fixed point in time. While this works in the concrete state space, PDB construction usually takes place in abstract space, imposed by an abstraction function that often projects some variables to don't cares. The automated selection of variables is important for its success but involved (Edelkamp 2001; 2007; Haslum et al. 2007; Kissmann and Edelkamp 2011).

Algorithmically, we start with the abstract goal set and iterate to successively compute the pre-image. Each state set in a layer is efficiently represented by a corresponding characteristic function. We may assume that the variable ordering is fixed and has been optimized prior to the search. For a given abstraction function the symbolic PDB $Heur(value, x)$ is initialized with the projected goal. As long as there are newly encountered states we take the current backward search frontier and generate the predecessor list with respect to the abstracted transition relation. Then we attach the current BFS level to the new states, merge them with the set of already reached states, and iterate. For non-unit action costs this process can be extended from breadth-first to cost-first levels, and it is possible to combine different symbolic heuristics by taking their maximum or by a controlled combination of their sum. The variables encoded in *value* are often queried at the bottom or at the top (in which case we obtain the equivalent to a vector of BDDs). For BDDA* it is more convenient to choose the one where the heuristic relation is partitioned into $Heur_0(x)$, ..., $Heur_k(x)$, with $Heur(value, x) = \bigvee_{i=0}^{k}(value = i) \wedge Heur_i(x)$.

BDDA* (Edelkamp and Reffel 1998), a.k.a. SetA* (Jensen, Veloso, and Bryant 2008), operates on a BDD priority queue $Open$. In case of discrete cost-values the $Open$ sets can be represented by BDDs. For the organization of the search that avoids BDD arithmetic, it is convenient to partition the state space. As we aim at *cost-optimal symbolic sequential planning*, the matrix-based version of BDDA* works on a partitioning of the search space in g- and h-values, where g is the cost of the path traversed so far and h is the heuristic estimate on the cost to reach the goal. To guarantee optimal cost, BDDA* expands this matrix along the f-diagonals with increasing g-values. The successors of the BDD $Open_{g,h}$ for a chosen transition with cost c are unified with the BDD $Open_{g+c,h'}$, where $h' \in \{0, \ldots, k\}$ is the partitioning obtained by the heuristic evaluation of the successor set.

Basic Improvements

Our starting point is the IPC 2011 version of the planner GAMER[2], described in (Kissmann and Edelkamp 2011). It applies symbolic PDB construction (for at most half the time) and BDDA* search (for the rest of the time) for non-unit cost domains or bidirectional BFS (for the full time) for unit cost domains. If backward search takes too long, abstractions are applied, otherwise a (partial) PDB in the concrete search space is constructed. In IPC 2011 GAMER did not score as well as it did in 2008. Of the twelve planners it finished ninth with only 148 solutions for the 280 instances (14 domains with 20 instances each), while one of the FAST DOWNWARD STONE SOUP versions (Helmert, Röger, and Karpas 2011) won with a total of 185 solutions. If we compare the number of solved instances of the domains with unit and with non-unit action costs the results are quite peculiar. In the four domains with unit action costs GAMER found only 23 solutions; only one participant was worse than that. In the remaining ten domains with non-unit action costs GAMER found 125 solutions; only three other planners were able to find more (with the maximum being 131). Based on the results of the competition we implemented some small improvements, which are as follows.

In the *solution reconstruction* for bidirectional BFS, GAMER supposed that at least one forward and at least one backward step were performed. The two easiest problems of VISITALL require only a single step, so that the solution reconstruction crashed.

In some cases, *parsing* the ground input took more than 15 minutes, so that actually no search whatsoever was performed in the domains with non-unit action costs. First it was parsed in order to generate a PDB, the calculation of which was killed after 15 minutes, and then the input was parsed again for BDDA*. In the domains with unit action costs it sometimes also dominated the overall runtime. Thus, we switched to a parser generator for Java programs, with which the parsing typically takes at most a few seconds.

In the most complex cases, generating the BDDs for the *transition relation* takes a long time, as well. The planner had to generate them twice in case of domains with non-unit action costs if it did not use the abstraction, once for the PDB generation and once for BDDA*. Instead, we store the transition relation BDDs, the BDD for the initial state and that for the goal condition on the hard disk; reading them from the disk is often a lot faster than generating them again from scratch.

In two of the new domains, namely PARKING and TIDYBOT, we found that the first backward step takes too long, often even more than 30 minutes, so that the decision whether to use bidirectional or unidirectional BFS could not be finished before the overall time ran out. In these cases, the single images for all the operators were quite fast, but the disjunction took very long. Thus, during the disjunction steps we entered the possibility to check whether too much time, in this case 30 seconds, has passed. If it has we *stop the disjunctions* and the planner only performs unidirectional BFS (or PDB generation using abstractions). This enabled us to find some solutions in TIDYBOT. The problem here is that the goal description allows for too many possibilities, because variables from only very few of the SAS$^+$ groups are present.

While the original planner used bidirectional BFS for domains with unit action costs, we tried running *BDDA* in all cases*, no matter if we are confronted with them or not. Thus, for the domains with unit action costs we treated all operators as if they had a cost of 1. We call this implementation *Matrix BDDA**.

[2]available at http://www.plg.inf.uc3m.es/ipc2011-deterministic

Algorithm 1 List-BDDA*.

Input: $\mathcal{A}$: finite set of operators.
$\mathcal{I}$: initial state.
$\mathcal{G}$: goal description.
$c : \mathcal{A} \mapsto \{1, \ldots, C\}$: action costs.
$Heur_h$: heuristic relation.
T_o: transition relation for operator $o \in \mathcal{A}$.
Output: cost-optimal plan.
$Open_0 \leftarrow \mathcal{I}$
for all $f = 0, \ldots$
  for all $g = 0, \ldots, f$
    $h \leftarrow f - g$
    $S \leftarrow Open_g \wedge Heur_h$
    if $(h = 0)$ **and** $(S \wedge \mathcal{G} \neq 0)$ **return** *ConstructSolution*
    for all $i = 1, \ldots, C$
      $Succ_i(x') \leftarrow \bigvee_{a \in \mathcal{A}, c(a)=i} \exists x \,.\, S(x) \wedge T_o(x, x')$
      $Open_{g+i} \leftarrow Open_{g+i} \vee Succ_i[x' \leftrightarrow x]$

List BDDA*

In Matrix BDDA*, all successors are classified according to their h-value by applying conjunctions with all the heuristic BDDs. When the number of heuristic values grows this can be inefficient, since some of these conjunctions could be avoided.

The representation in the matrix can be simplified to the vector for the states in the *Open* list ordered along the g-value. The reasoning behind this strategy is to defer the heuristic calculation by computing the conjunction of the successor set with the heuristic estimate only when it is needed for expansion in the currently traversed f-diagonal.

The pseudo-code of the resulting algorithm *List BDDA** for non-zero cost operators is shown in Algorithm 1. All inputs to the algorithm $\mathcal{A}, \mathcal{I}, \mathcal{G}, c, Heur_h, T_o$ are represented as BDDs.

It is simple to add duplicate detection to the algorithm using another set *Closed* for the set of expanded states. The handling of zero-cost operators adds another BFS loop to the code as these operators are to be preferred in the exploration. While Matrix BDDA* uses BFS to get the states reachable with zero-cost operators independent of their actual h values, in the list version we apply a conjunction with the heuristic value to get only those states in the current f-diagonal.

Symbolic Merge-and-Shrink

It is not difficult to observe that the precomputed memory structure of the M&S heuristic can be cast as a symbolic representation of an integer-valued function. This function can be extracted in form of an ADD (Bahar et al. 1997) allowing to enrich a symbolic heuristic planner to exploit this expressive estimate. The precomputed ADD is converted to a vector of BDDs (Bryant 1985) and can be plugged into an optimal symbolic heuristic search planner. Such an approach is general to all M&S variants using a linear merge strategy, including the latest improvements based on bisimulation reductions (Nissim, Hoffmann, and Helmert 2011).

Every intermediate abstraction corresponds to a layer in the ADD and each abstract state corresponds to an ADD node. When a new variable is merged into the abstraction, every state is split into k states in the next level, one for each value of the variable. ADD nodes representing the parent state are connected with the nodes representing its successors. As M&S works with finite domain variables but the ADD is defined for binary variables, each node with k successors is converted to a binary tree with $\log_2 k$ layers.

To compute the ADD of an M&S heuristic we start by generating the sink nodes associated with the different heuristic values of the abstract states in the last layer. Then, recursively, nodes in the previous layer can be constructed pointing to the nodes in layers already computed. During the ADD construction we ensure the application of the reduction rules, so that the size of the final ADD is usually smaller than the M&S heuristic structure.

ADD Complexity

The symbolic ADD representation of the M&S heuristic can be computed in time and space $O(nM)$, where n is the size of the Boolean state vector and M is the pre-defined maximum number of states. Moreover, the representation of the heuristic as a sequence of BDDs $h_0, \ldots, h_{\max}$ can be computed in time and space $O(h_{\max} nM)$. The time and space complexities are implied by the maximum sizes of the state spaces for the construction of the next-variables tables in the explicit search construction of the M&S heuristic. BDD reduction is a linear time operation (Sieling and Wegener 1993) and only decreases the size.

The ADD sizes for the two M&S heuristics are shown in Table 1. For each domain we provide the number of instances in which the heuristic computation was finished in 30 minutes as well as the sizes of the largest ADD for each domain. Surprisingly the ADDs are small, especially for the greedy version of M&S, showing that not much memory is spent once the ADD has been computed.

Limits and Possibilities

In Proposition 2.1 in (Nissim, Hoffmann, and Helmert 2011) referring to (Helmert, Haslum, and Hoffmann 2007) it is said that the M&S heuristic strictly generalizes the PDB heuristic, as with only merging variables by computing their synchronous product all pattern database heuristics based on projecting the variables can be constructed. Moreover, (Nissim, Hoffmann, and Helmert 2011) states that M&S can compute perfect heuristics in polynomial time, where PDBs cannot. The distinguishing example is the GRIPPER domain.

In a symbolic setting, this reasoning, however, is no longer immediate. If all the variables are included in the pattern, the original state space can be fully traversed resulting in the optimal heuristic. As shown in (Edelkamp and Kissmann 2008), the BDD exploration that computes the perfect heuristic in GRIPPER is polynomial. Thus, even though it is easier to relax M&S for computing non-perfect heuristics (Katz, Hoffmann, and Helmert 2012), as the representational power of both alternatives is equivalent (an ADD

Table 1: Number i of instances with M&S heuristic and maximum number n of ADD nodes over all instances for all domains of the sequential optimal track of IPC 2011.

	M&S (gop')		M&S (gop)	
Problem	i	n	i	n
NoMystery	20	197,445	20	915
Parking	0	—	20	2,260
Tidybot	0	—	20	15
VisitAll	20	3,225,813	20	7,381
Barman	20	177,294	20	45
Elevators	20	62,594	0	—
Floortile	20	1,278,950	8	6,283
Openstacks	20	102,486	4	134,780
PARC-Printer	19	4,606,533	20	11,788
Peg-Solitaire	20	42,170	0	—
Scanalyzer	16	356,698	6	29,921
Sokoban	20	1,339	1	33,525
Transport	20	257,898	20	753
Woodworking	20	248,263	20	439,489

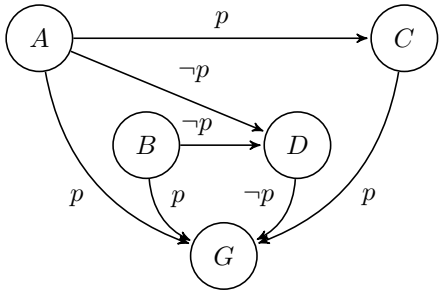


Figure 1: Example of bisimulation. A, B, C, D and G are states in one level of the M&S process, while p and $\neg p$ are variable assignments that serve as a precondition of the according operators.

for M&S and a list of BDDs for the symbolic PDB) both approaches can potentially derive optimal heuristics in the same domains.

However, even if the M&S bisimulation gets the perfect heuristic, it does not always result in a reduced representation of the ADD. First, we observe that for any (e. g., the perfect) heuristic – no matter how it is computed – by the uniqueness property, the according ADDs (following the same variable ordering) have to be the same. Secondly, we can construct an intuitive example, where shrinking is not able to compute the most reduced form of the heuristic.

Figure 1 shows an example where there are not any bisimilar states. The transition labels have already been reduced so that they refer to variables that have not yet been merged. In the example these labels are preconditions and they only refer to a binary variable p. All the transitions have unit cost and the goal is to reach state G.

We say that two abstract states s and s' are *equivalent*, if and only if, for every value assignment to the variables that have not yet been merged the goal distance remains the same. If two abstract states are equivalent, their corresponding ADD nodes can be unified according to the ADD reduction rule (2). It is easy to see that states A and B in the example are equivalent because in case that p holds both have a cost of 1, while if $\neg p$ holds both have a cost of two. However, they are not bisimilar because B does not have any transition to state C. Obviously, states C and D are not bisimilar, given that their transitions have different labels. Therefore, no pair of states is reduced by bisimulation.

However, since in the end only the distance to the goal matters, those transitions that are not part of an optimal path should not be taken into account. In the example, if the transition $A \to C$ is not necessary then A and B are equivalent. Checking if a transition is necessary in any optimal path is not trivial as it needs to consider all the combinations of values of the variables that have not been merged.

It is possible to extend the example by adding an exponential number of equivalent states that are not bisimilar because they have different transitions that are not needed by any of their optimal paths. Therefore, this can cause an exponential gap between the size of the intermediate abstraction and the final reduced heuristic.

On the other hand, symbolic backward search iteratively constructs the reduced BDDs for every cost. In the absence of zero-cost operators, the intermediate BDDs are always fully reduced. Thus, in some domains the size of the BDDs used by symbolic partial PDBs may be exponentially smaller than the M&S representation. The counterpart is that these BDDs are computed with images of the transition relation, which in some domains may be expensive.

Experiments

We use two planners as the basis for our experiments, namely Fast Downward[3] (FD) (Helmert 2006) offering the M&S heuristic for explicit-state planning, and Gamer for executing symbolic heuristic search. Both systems include their latest improvements.

The software infrastructure is taken from the resources of the International Planning Competition IPC 2011. We implemented the proposed refinements in Gamer (Matrix BDDA* and List BDDA*) using the CUDD library of Fabio Somenzi (compiled for 64-bit Linux using the GNU gcc compiler, optimization option -O3). For the experiments we used our own machine (Intel Core $i7$ 920 CPU with 2.67 GHz and 24 GB RAM) with the same settings concerning timeout (30 minutes) and maximal memory usage (6 GB) as in the competition. While the time and memory settings are the same as in the competition, other parameters of the computer are different.

We did experiments with two different configurations of the M&S heuristic, one using only greedy bisimulation (gop) and one using DFP-gop (gop') with parameter $M = 200{,}000$. The M&S planner presented in the competition used these two strategies serially, first the version with gop for 800 seconds and afterward the one with DFP-gop for 1,000 seconds. We decided to run both parts independently to see how well we perform against a more tradi-

[3]Retrieved on February 22nd 2012 from the Fast Downward repository at `http://hg.fast-downward.org`.

tional, i. e., non-portfolio, planner as well. The pattern selection for symbolic PDBs is the same as used by GAMER in the competition (Kissmann and Edelkamp 2011).

We look at two different variable ordering strategies used by the competition planners, the one applied in FD and the one applied in GAMER. The FAST DOWNWARD ordering looks at strongly connected components and weights of the causal graph (Helmert 2006). Highly related variables are pushed to the top and goal variables are pushed to the bottom of the ordering. The GAMER ordering also looks at the dependency of variables and is the result of a random local search to improve the ordering according to incrementally computing the optimization function $\sum_{1\leq i,j,\leq n,(u_i,v_j)\in D}(\pi(i)-\pi(j))^2$, where π denotes the applied permutation and D denotes the set of the causal dependencies. Thus, highly related variables are pushed to the middle of the ordering.

The results are shown in Table 2. All the small improvements in Matrix A* compared to GAMER helped mainly in the domains with unit action costs. There we are now able to find the two trivial solutions in VISITALL, in PARKING we find one solution and in TIDYBOT we find eight solutions – in both domains we failed completely in the competition. In the domains with non-unit action costs the new parser helped us to find three additional solutions in the SCANALYZER domain. Overall, Matrix BDDA* solves 158 problems, which is 12 problems more than the competition version of GAMER run on our machine.

When comparing both implementations of BDDA*, List BDDA* is better than Matrix BDDA* when the M&S heuristic is used or when PDBs are used in FAST DOWNWARD ordering; in case of PDBs and GAMER ordering both find the same number of solutions. On the other hand, explicit A* beats both BDDA* versions when using FAST DOWNWARD ordering and one of the M&S heuristics, while with GAMER ordering it is better with the gop M&S heuristic but worse with the gop' heuristic. With FAST DOWNWARD ordering and the PDB heuristic both BDDA* versions are better than explicit A*, while with GAMER ordering there is no difference in the total number of found solutions.

The symbolic PDB heuristic did not use abstraction in most of the domains. With GAMER ordering abstraction is used in some problems of PARC-PRINTER, PARKING, SOKOBAN and TIDYBOT. With FAST DOWNWARD ordering abstraction is also used in FLOORTILE and OPENSTACKS. In all the other domains the heuristics were computed by symbolic backward search until all states had been reached or the time limit of 15 minutes had been expired. The perfect heuristic was found for 70 problems when using GAMER ordering and for 60 with FAST DOWNWARD ordering.

The results are highly influenced by the choice of the variable ordering. Overall the FAST DOWNWARD ordering is better for the M&S heuristic, while GAMER's ordering helps the symbolic exploration. Due to this, the integration of symbolic search and the M&S heuristic is difficult because both have to use the same ordering.

The variable ordering matters not only for the kind of planner used but also for the domain it is used on. For example, in NOMYSTERY, PARKING, VISITALL, FLOORTILE, PARC-PRINTER, and SCANALYZER the FAST DOWNWARD ordering is better in most cases for all planners, while in ELEVATORS, OPENSTACKS, SOKOBAN, and WOODWORKING the GAMER ordering is the better choice.

Overall, the best choice is to use any of the three planners with PDBs and GAMER ordering. However, as the M&S heuristic takes less time to compute it is more suitable to be run more than once. Using the same configuration as in the competition, FD with the M&S heuristic can solve 171 problems, 2 more than in the competition, probably due to some bugfixes and performance boosts in that planner as well. On our machine, neither of the two versions took more than 600 seconds for any of the solved instances, so that a combination of both really is reasonable. The memory limit of 6 GB is what prevents them from finding more solutions.

Given a perfect oracle that tells us for each domain which heuristic, which ordering, and which version of BDDA* to use we would be able to find 185 solutions. In order to come up with such an oracle we have to investigate the differences between the planners, heuristics and orderings further and find out why some work better than others in certain domains.

Conclusion

According to the outcome of the last two IPCs, heuristic and symbolic search are two leading methods for sequential optimal planning.

In this paper we have seen a head-to-head race of two symbolic high-quality estimates, namely the PDB and the M&S heuristic. Surprisingly, the former won (if we stick to a single planner run). With PDB search, we arrive at 158 solutions in the set of competition instances (30 to 32 in the domains with unit action costs and 126 to 128 in those with non-unit action costs, dependent on the actual planner used). If we would use Matrix BDDA* for the unit cost problems and List BDDA* for the problems with non-unit costs we arrive at 160 problems being solved. With this we are still behind the number of 171 problems solved by explicit search with two different versions of the M&S heuristic. Nevertheless, the performance of solving 128 instances with non-unit costs is closing the gap to the state-of-the-art. With 131 found solutions only one of the FAST DOWNWARD STONE SOUP planners is better. Moreover, if we were to exclude the PARC-PRINTER domain, which is special due to the extremely high and diverse action costs, the picture would actually be fortunate for us.

The variable ordering for the M&S heuristic influences both the quality of the estimate and the symbolic exploration. The heuristic choice applied in FD pleases the M&S heuristic, while the optimization applied in GAMER pleases symbolic exploration. Future work is needed to combine the two for a competitive BDDA* exploration with the M&S heuristic.

The small ADD sizes for the M&S heuristic documented in this paper suggest that there is sufficient memory for computing the maximum of more than one heuristic (in ADD

Table 2: Number of solved problems for all domains of the sequential optimal track of IPC 2011.

Domain	FAST DOWNWARD Ordering									GAMER Ordering								
	Explicit A*			Matrix BDDA*			List BDDA*			Explicit A*			Matrix BDDA*			List BDDA*		
	gop	gop'	PDB	gop	gop'	PDB	gop	gop'	PDB	gop	gop'	PDB	gop	gop'	PDB	gop	gop'	PDB
NOMYSTERY	13	**20**	14	14	**20**	16	15	**20**	16	13	12	12	14	17	14	13	18	13
PARKING	**7**	0	0	3	0	0	3	0	0	0	0	0	0	0	1	0	0	1
TIDYBOT	**13**	0	12	6	0	6	6	0	6	**13**	0	8	9	0	6	6	0	5
VISITALL	13	11	10	5	5	12	12	12	12	11	9	11	11	10	11	11	10	11
Total (unit cost)	**46**	31	36	28	25	34	36	32	34	37	21	31	34	27	32	30	28	30
BARMAN	4	4	4	4	4	4	4	4	4	4	4	4	**6**	5	4	4	4	4
ELEVATORS	0	11	18	0	16	**19**	0	16	**19**	6	12	**19**	6	14	**19**	5	17	**19**
FLOORTILE	3	7	**12**	3	7	**12**	3	7	**12**	3	4	11	3	4	8	3	4	8
OPENSTACKS	4	16	15	4	15	14	4	15	15	5	16	16	4	**20**	**20**	5	**20**	**20**
PARC-PRINTER	11	**14**	10	8	11	7	10	11	9	11	12	10	8	9	7	10	7	8
PEG-SOLITAIRE	0	**19**	**19**	0	**19**	**19**	0	**19**	**19**	0	**19**	**19**	0	**19**	17	0	**19**	17
SCANALYZER	6	**10**	9	6	9	9	6	9	9	3	8	9	3	8	9	3	7	9
SOKOBAN	1	**20**	4	1	13	12	1	12	12	3	**20**	17	2	18	19	2	18	19
TRANSPORT	6	7	9	6	7	9	6	7	**10**	6	6	9	6	6	7	6	6	8
WOODWORKING	9	6	7	10	7	5	10	7	7	9	9	13	6	8	**16**	13	12	**16**
Total (others)	44	114	107	42	108	110	44	107	116	50	110	127	44	111	126	51	114	**128**
Total (all)	90	145	143	70	133	144	80	139	150	87	131	**158**	78	138	**158**	81	142	**158**

representation). This results in a consistent, strictly more informed heuristic for the (BDD)A* exploration and provides a way of combining the accuracy of PDBs and M&S heuristics.

In many instances that are solved by BDDA* with PDBs no abstraction is applied, meaning that blind symbolic backward search in the concrete state space is either finalized or truncated by the time limit. As a consequence at least in domains where backward search does not explode immediately (due to illegal states produced), bidirectional blind symbolic search is best.

Another competitor to look at would be the planning heuristic h^+. In a restricted setting it has been compiled into a logic program and to a d-DNNF, where d-DNNFs are another succinct representation for Boolean functions (Bonet and Geffner 2008).

Acknowledgments

Thanks to Deutsche Forschungsgesellschaft (DFG) for support in the project ED 74/11-1. Thanks to the Spanish Government for support in the MICINN projects TIN2011-27652-C03-02, TIN2008-06701-C03-03 and to the Comunidad de Madrid for support in the project CCG10-UC3M/TIC-5597.

References

Anderson, K.; Schaeffer, J.; and Holte, R. C. 2007. Partial pattern databases. In *SARA*, 20–34.

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

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.

Ball, M., and Holte, R. C. 2008. The compression power of symbolic pattern databases. In *ICAPS*, 2–11.

Bonet, B., and Geffner, H. 2008. Heuristics for planning with penalties and rewards formulated in logic and computed through circuits. *Artif. Intell.* 172(12–13):1579–1604.

Bryant, R. E. 1985. Symbolic manipulation of boolean functions using a graphical representation. In *DAC*, 688–694.

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

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2006. Directed model checking with distance-preserving abstractions. In *SPIN*, 19–36.

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2009. Directed model checking with distance-preserving abstractions. *STTT* 11(1):27–37.

Edelkamp, S., and Kissmann, P. 2008. Limits and possibilities of BDDs in state space search. In *AAAI*, 1452–1453.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In Herzog, O., and Günter, A., eds., *KI*, volume 1504 of *Lecture Notes in Computer Science*, 81–92. Springer.

Edelkamp, S. 2001. Planning with pattern databases. In *ICAPS*, 13–34.

Edelkamp, S. 2005. External symbolic heuristic search with pattern databases. In *ICAPS*, 51–60.

Edelkamp, S. 2007. Automated creation of pattern database search heuristics. In *MOCHART*, 35–50.

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*, 1007–1012.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*, 162–169.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In Boddy, M. S.; Fox, M.; and Thiébaux, S., eds., *ICAPS*, 176–183. AAAI.

Helmert, M.; Haslum, P.; and Hoffmann, J. 2008. Explicit-state abstraction: A new method for generating heuristic functions. In *AAAI*, 1547–1550.

Helmert, M.; Röger, G.; and Karpas, E. 2011. Fast downward stone soup: A baseline for building planner portfolios. In *ICAPS-Workshop on Planning and Learning (PAL)*.

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

Jensen, R. M.; Veloso, M. M.; and Bryant, R. E. 2008. State-set branching: Leveraging BDDs for heuristic search. *Artif. Intell.* 172(2–3):103–139.

Katz, M., and Domshlak, C. 2010. Implicit abstraction heuristics. *J. Artif. Intell. Res. (JAIR)* 39:51–126.

Katz, M.; Hoffmann, J.; and Helmert, M. 2012. How to relax a bisimulation? In *ICAPS*. To Appear.

Kissmann, P., and Edelkamp, S. 2011. Improving cost-optimal domain-independent symbolic planning. In *AAAI*, 902–907.

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*, 1983–1990.

Pang, B., and Holte, R. C. 2011. State-set search. In *SOCS*.

Sieling, D., and Wegener, I. 1993. Reduction of OBDDs in linear time. *Inf. Process. Lett.* 48(3):139–144.

Wegener, I. 2000. *Branching Programs and Decision Diagrams*. SIAM.

Making Reasonable Assumptions to Plan with Incomplete Information

Sammy Davis-Mendelow
Department of Computer Science
University of Toronto
Toronto, Canada

Jorge A. Baier
Depto. de Ciencia de la Computación
Pontificia Universidad Católica de Chile
Santiago, Chile

Sheila A. McIlraith
Department of Computer Science
University of Toronto
Toronto, Canada

Abstract

Many practical planning problems necessitate the generation of a plan under incomplete information about the state of the world. In this paper we propose the notion of Assumption-Based Planning. Unlike conformant planning, which attempts to find a plan under all possible completions of the initial state, an assumption-based plan supports the assertion of additional assumptions about the state of the world, simplifying the planning problem. In many practical settings, such plans can be of higher quality than conformant plans. We formalize the notion of assumption-based planning, establishing a relationship between assumption-based and conformant planning, and prove properties of such plans. We further provide for the scenario where some assumptions are more preferred than others. Exploiting the correspondence with conformant planning, we propose a means of computing assumption-based plans via a translation to classical planning. Our translation is an extension of the popular approach proposed by Palacios and Geffner and realized in their T0 planner. We have implemented our planner, A0, as a variant of T0 and tested it on a number of expository domains drawn from the International Planning Competition. Our results illustrate the utility of this new planning paradigm.

Introduction

In many real-world planning problems, only a subset of the state of the world may be known. Conformant planning, conditional planning, probabilistic planning and contingent planning are among the approaches used to address such planning scenarios. Whereas classical planning assumes complete information about the state of the world, conformant planning assumes incomplete information but necessitates generation of a plan that relies only on what is known. This makes planning difficult and can lead to poor plans.

In this paper we define the notion of *assumption-based planning*. Assumption-based planning attempts to find a middle-ground between conformant and classical planning wherein the planner dynamically asserts reasonable, calculated assumptions about the uncertainty in the world in order to generate a valid plan given those assumptions. In contrast to contingent planning that exploits strategic sensing to resolve uncertainty, assumption-based planning is well-suited to scenarios where resolving uncertainty directly is impossible, difficult, or expensive. Consider the simple task of planning your trip home at the end of a work day. You don't know for certain that the subway will be running, you have no way of finding out, but it's reasonable to assume so. Making this assumption supports generation of a reasonable plan. The conformant plan might have you walking home!

The term assumption-based planning has been coined for a number of diverse planning activities that broadly relate to assumptions. Albore and Bertoli (2004; 2006) used the term to describe a notion of planning in which an assumption is provided a priori as a linear temporal logic (LTL) formula and a plan is generated predicated on this assumption. Their work is more closely related to planning with LTL domain control knowledge (e.g., Bacchus and Kabanza 2000). Pellier and Fiorino (2004; 2005) also use the term to describe a multi-agent approach to devising a shared global plan via a conjecture/refutation cycle, where agents exchange proposals and counter-proposals in an argumentation dialogue. Their basic formulation shares commonalities with our approach, but the approach to plan generation is fundamentally different and the two pieces of work were developed independently.

Here, assumption-based planning is related to the characterization of abduction as theory formation (e.g., Poole, Goebel, and Aleliunas 1987) wherein additional facts about the world are conjectured in order to explain an observation. It is also somewhat related to the notion of generating explanations for dynamical systems (e.g., Sohrabi, Baier, and McIlraith 2011). Indeed, Reiter and de Kleer (1987) established the relationship between abduction (explanation generation) and assumption-based reasoning for static systems, and Conrad and Williams (2011) employed aspects of assumption-based truth maintenance in their Drake executive for temporal plans. In (Göbelbecker, Gretton, and Dearden 2011; Bonet and Geffner 2011; Albore and Geffner 2009), contingent planners may make assumptions that can be verified through sensing later on.

In contrast to previous work, we provide a formal characterization of assumption-based planning establishing a correspondence to conformant planning. Exploiting this correspondence, we provide a translation of an assumption-based planning problem to a classical planning problem. , building on the popular translation developed by Palacios and Geffner (2009). We prove the soundness and completeness of our translation. This provides us with a means of generating assumption-based plans using classical

planners.

We also argue for the merit of preferred assumption-based plans and propose a means of realizing such plans via cost-based planning. We implement these two approaches and perform experiments to illustrate their viability and assess some of their properties.

Beyond planning, the assumption-based planning paradigm has compelling applications in a diversity of application areas including diagnosis and verification.

Characterization

In this section we formally define assumption-based planning and initial state assumption-based planning, as well as state the equivalence of the two given deterministic actions and no exogenous events.

Background

We now define the fundamental notions that will be used in the rest of the paper. Our first definition is for *planning problems*. [1]

Definition 1 (Planning Problem) *A planning problem is a tuple $P = (F, O, I, G)$ where F is a finite set of fluent symbols, O is a finite set of action operators, I is a set of clauses over F, defining the set of possible initial states, and G is a boolean formula over symbols in F, that defines a goal condition.*

Every action $a \in O$ is defined by a conjunction of fluent literals, $prec(a)$ (preconditions) and a set of conditional effects $C \rightarrow L$ where L is a fluent literal that is made true when the action is executed and the conjunction of fluent literals C holds. An unconditional effect is one in which $C = \mathsf{true}$; L is made true every time the action is executed.

Example 1 Consider a car-driving domain in which a car can drive between locations. There are three actions: $drive(x, y)$, $turnOn$, and $turnOff$. There are only two locations: $home$ and $office$. Initially we only know that the car is at home and that its engine is not on. Hence, the initial state I is given by $\{at(home), \neg at(office), \neg engineOn\}$. Action $turnOn$ turns the engine on if the battery is working, represented by the conditional effect $\{batteryOk \rightarrow engineOn\}$, and has no preconditions (i.e., $prec(a) = \mathsf{true}$). Action $drive(x, y)$ requires as precondition that $at(x)$ and that $engineOn$, and has the (unconditional) effects $at(y)$, $\neg at(x)$. Action *fixBattery* has no precondition and a single conditional effect $\neg batteryOk \rightarrow batteryOk$. Finally, the objective is to bring the car to the office: $G = at(office)$.

A *planning state* s is defined by a set of fluent symbols, which represent all that is true. Each system state s induces a propositional valuation $M_s : F \rightarrow \{\mathsf{true}, \mathsf{false}\}$ that maps any fluent literal in s to true, and all other literals to false.

We say a state s is *consistent* with a set of clauses $\mathcal{C}$, if $M_s \models c$, for every $c \in \mathcal{C}$. Intuitively, $M_s \models \phi$ stands for "boolean formula ϕ holds true in state s".

[1] In the planning literature, the following definition is usually used for *conformant* planning problems. Here we do not wish to cast a planning problem as conformant a priori. As we will see, this definition generalizes also to our assumption-based paradigm.

An action a is *executable* in a state s if $M_s \models prec(a)$. If a is executable in a state s, we define its successor state as $\delta(a, s) = (s \setminus Del) \cup Add$, where Add contains a fluent f iff $C \rightarrow f$ is an effect of a and $M_s \models C$. On the other hand Del contains a fluent f iff $C \rightarrow \neg f$ is an effect of a, and $M_s \models C$. We define $\delta(a_0 a_1 \dots a_n, s) = \delta(a_1 \dots a_n, \delta(a_0, s))$, and $\delta(\epsilon, s) = s$. A sequence of actions α is *executable* in s if $\delta(\alpha, s)$ is defined. Furthermore α is executable in P iff it is executable in s, for any s consistent with I.

Below we define the notion of *execution trace* which intuitively characterizes maximal state trajectories that could result from the execution of an action sequence when performed in some of the possible initial states of a planning problem.

Definition 2 (Execution Trace) *A sequence of planning states $\sigma = s_0 s_1 \cdots s_k$ is an* execution trace *of $\alpha = a_0 a_1 \dots a_n$ in planning problem $P = (F, O, I, G)$ iff (1) s_0 is consistent with I, (2) $\delta(a_i, s_i) = s_{i+1}$, for all $i < k$, and (3) either $k = n + 1$ or $k < n + 1$ and $\delta(s_k, a_k)$ is undefined.*

Definition 3 (Successful Execution Trace) *An execution trace σ for α is successful iff $|\sigma| = |\alpha| + 1$.*

Naturally, we are interested in execution traces that lead to goal satisfaction, i.e., for which the goal holds in the final state of the sequence of planning states. Formally,

Definition 4 (Leads to) *An execution trace $\sigma = s_0 \cdots s_k$ leads to (goal formula) G, iff $M_{s_k} \models G$.*

With the previous definitions in hand, we define the standard notion of conformant plan.

Definition 5 (Conformant Plan) *A sequence of actions α is a conformant plan for $P = (F, O, I, G)$ iff every execution trace of α is successful and leads to G.*

In our car-driving example, the sequence $fixBattery; turnOn; drive(home, office)$ is a conformant plan. The reader can easily verify that action *fixBattery* is needed in any conformant plan, since two planning states are consistent with I: $\{at(home), batteryOk, \neg engineOn\}$, and $\{at(home), \neg batteryOk, \neg engineOn\}$.

Assumption-Based Planning

Consider an extension of our car-driving example where, in addition to the battery, many other car components are modeled as potentially malfunctioning. In the absence of information regarding the status of each component, a conformant plan would require fixing each component, which would result in either a long, poor quality plan, or possibly no plan at all. A contingent plan could be similarly poor, requiring significant computation or contingencies for unlikely scenarios. Instead we would like the planning system to simplify the task, as people do, by automatically assuming, in the absence of evidence to the contrary, that the battery and other components are functioning correctly in the initial state. Later on, we would like the planner to be capable of assuming that the freeway is not blocked in the state immediately before entering it. In general, we would like the planner to be able to make reasonable assumptions about *any* state along the execution.

Given a planning problem with an incomplete initial state, the task of assumption-based planning requires computing two elements: (1) a set of assumptions that are made at different states during the execution of the plan, and (2) a sequence of actions that, given the assumptions, is guaranteed to reach the goal. As such, the main difference between assumption-based planning and conformant planning is related to the computation of assumptions in addition to the computation of a plan.

Formally an *assumption-based planning problem* is a tuple $P = (F, O, I, G, T)$, where F, O, I, and G are defined exactly as for regular planning problems (Definition 1), and T is a subset of F and denotes the set of *assumable fluents*. T may be equivalent to the set of domain fluents or it may be restricted to an application-specific subset, such as the set of fluents corresponding to the normal functioning of car components in our car example. An assumption-based plan is a pair (ρ, α) where $\alpha = a_0 \cdots a_k$ is a sequence of actions and $\rho = h_0 \cdots h_{k+1}$ is a sequence of boolean formulae. Each h_i is a boolean formula over the fluents in T and represents assumptions made about the i-th state visited when performing α.

The execution traces that we will be interested in are those that *conform to* ρ; i.e., are such that they are consistent with the assumptions. Formally,

Definition 6 (Conforms to) *An execution trace $\sigma = s_1 \cdots s_k$ conforms to a sequence of boolean formulae $\rho = h_1 \cdots h_n$ with $k \leq n$ iff $M_{s_i} \models h_i$, for every $i \in \{1, \ldots, k\}$.*

Finally, each of the execution traces of α that conform to ρ must actually lead to the goal. A formal definition of an assumption-based plan follows.

Definition 7 (Assumption-Based Plan) *The pair (ρ, α), where α is a sequence of k actions, and ρ is a sequence of $k+1$ boolean formulae over T is an assumption-based plan for $P = (F, O, I, G, T)$ iff any execution trace of α that conforms to ρ is successful and leads to G, and furthermore at least one such execution trace exists.*

Intuitively for every consistent completion of the initial state the execution trace is either successful and leads to the goal or is pruned by ρ. Note that by the definition of *conforms to*, if an execution is not successful and doesn't conform to ρ, then it is pruned before the terminating state. Assumption-based planning may be reduced to a conformant plan when all assumptions are restricted to be trivial. An assumption is trivial when it is entailed by the state in which it is made. Trivial assumptions may be forced by restricting the set T, the extreme case being when T is empty.

In our car example, $\pi = (\rho, \alpha)$, with $\rho = batteryOk; \mathsf{true}; \mathsf{true}$, $\alpha = turnOn; drive(home, office)$ is an assumption-based plan that assumes the battery is initially working and uses two actions to achieve the goal.

An important fact to note at this point is that, given the current definition of the problem, assumptions may in some cases provide too much flexibility. In fact, one can imagine cases in which we assume the preconditions of an action that achieves the goal or even simply assume that the goal is true in the initial state. Of course, this could lead to over-simplified solutions, which may not be desirable. This issue can be tackled by defining a notion of quality over assumption-based plans. We discuss this in more detail in the preferred assumption-based planning section.

Relation to Abduction The notion of assumption-based planning draws some intuition from the notion of abduction and in particular from the logical characterization of abduction as *theory formation* (e.g., Poole, Goebel, and Aleliunas 1987; Console, Dupre, and Torasso 1989). Informally, given an observation, abduction is inference to the best explanation for that observation. More formally, given a background theory Σ, a distinguished set of *abducible* literals $\mathcal{E}$, and an observation formula O, the formula E constructed from $\mathcal{E}$ is an abductive explanation for O iff (i) $\Sigma \not\models O$, (ii) $\Sigma \cup E$ is satisfiable, and (iii) $\Sigma \cup E \models O$. The definition of *best* abductive explanation is often application specific and imposes further criteria on E. Note that the designation of a distinguished set of literals, T, from which assumptions are drawn is analogous to the abducibles used to characterize the notion of abductive inference, as described above. This set is domain specific and is used to restrict assumptions to those that are reasonable for the domain. For example, in automated diagnosis, the abducibles are restricted to literals that designate the malfunctioning of different system components – the building blocks of diagnoses.

Initial-State Assumption-Based Planning

In many settings it is convenient or sufficient to make assumptions only about the initial state of the world. In other words, to make $h_i = \mathsf{true}$ for every $i > 0$. We call this class of problems initial-state assumption-based planning. An initial-state assumption-based plan is denoted by (h_0, α), where h_0 is a boolean formula over T that corresponds to an assumption made on the initial state.

The formal relation between conformant planning and initial-state assumption-based planning is straightforward, and is established in the following proposition.

Proposition 1 *The tuple (h_0, α) is an initial-state assumption-based plan for planning problem $P = (F, O, I, G, T)$ iff α is a conformant plan for $P' = (F, O, I \cup h_0, G)$.*

Note that this proposition *does not* imply that an assumption-based plan can be directly computed using a conformant planner, since a conformant planner is not able to compute assumptions. In addition, a relation between assumption-based planning and initial-state assumption-based planning can be established.

Theorem 1 *If $P = (F, O, I, G, T)$ and (ρ, α) is an assumption-based plan for P, then there exists an h_0 and a T' such that (h_0, α) is an initial-state assumption-based plan for $P' = (F, O, I, G, T')$. Furthermore, h_0 can be computed from ρ, P and α in time $2^{\mathcal{O}(|\alpha|)}$.*

Proof sketch: By applying regression rewriting (Waldinger 1977) to the sequence of assumptions, ρ, regressing them over the plan α we can obtain a formula ϕ_G that corresponds to the conditions under which α is executable in the initial state. From ϕ_G we can extract h_0, which must be consistent with I but not entailed by I. This regression may require assumptions in the initial state of fluents not in T, but this

will not affect which execution traces succeed and lead to G. Regression is worst-case exponential in $|\alpha|$, but is linear in $|\alpha|$ if there are no actions with conditional effects in α. □

As a consequence of this theorem, if α is a sequence of actions for which there is some ρ such that (ρ, α) is an assumption-based plan, then we can construct an assumption h_0 on the initial state using α such that (h_0, α) is an assumption-based plan. The proof of the above theorem (omitted here for space) actually gives a constructive algorithm for h_0 that relies on regressing ρ over α.

The proof of Theorem 1 establishes a means of constructing an initial-state assumption-based plan (h_0, α) from an arbitrary assumption-based plan (ρ, α) via regression. Under certain conditions, one can similarly construct an arbitrary assumption-based plan (ρ, α) from an initial-state assumption-based plan (h_0, α) by *progressing* aspects of h_0 (Lin and Reiter 1997). Intuitively, this provides a means of generating an assumption-based plan that makes assumptions at the point at which they are needed, and no sooner. This has particular value with respect to monitoring the execution of a plan, and may be used in settings in which exogenous events may occur.

We now analyze aspects that relate to the complexity of assumption-based planning. As it turns out, the definition of assumption-based planning is general enough that its complexity seems to lie across a spectrum of complexity classes, depending on which literals are allowed to be assumed. Below we provide two complexity results showing that assumption-based planning is complete for two complexity classes. Our first result follows directly from the fact that conformant planning is EXPSPACE-complete (Haslum and Jonsson 1999).

Theorem 2 *Given an assumption-based planning problem $P = (F, O, I, G, T)$, where T contains no fluents mentioned in non-unary clauses of I, deciding whether or not an assumption-based plan exists is EXPSPACE-complete.*

However, as more information can be assumed, the complexity of the decision problem move down to that of classical planning.

Theorem 3 *Given an assumption-based planning problem $P = (F, O, I, G, T)$, where T contains all fluents mentioned in non-unary clauses of I, deciding whether or not an assumption-based plan exists is PSPACE-complete.*

Proof sketch: For membership, we propose the following NPSPACE algorithm: guess the assumptions h_0 such that $I \cup h_0$ has a unique model, then call a PSPACE algorithm (like the one suggested by de Giacomo and Vardi (1999)) to decide (classical) plan existence. Then we use the fact that NPSPACE=PSPACE. Hardness is given by the fact that classical planning, a PSPACE-complete problem (Bylander 1994), can be straightforwardly reduced to assumption-based planning. □

Naive Approach to Assumption-Based Planning

Theorem 3 implies that when the set of assumable fluents contain all fluents appearing in non-unary clauses of I, assumption-based planning can be reduced to classical planning. Indeed, a naive algorithm for this type of assumption-based planning can be proposed by building a classical planning problem in which the planner first has to "guess" an assumption on the initial state, and then find a sequence of actions.

Specifically, the classical problem P' is like P but augmented with additional actions that can only be performed at the initial state and have as an objective to generate an initial state consistent with I. There is an exponential number of these actions. If $a_0 a_1 \dots a_n$ is a plan for P, we construct the initial-state assumption-based plan α as follows. h_0 is constructed with the facts true in the state s that a_0 generates. a is simply set to $a_1 \dots a_n$. Of course, the approach derived by the proof of this theorem is very impractical as it grows the size of the problem exponentially. In an extended version of our car example, in which we have n components of the car whose state is unknown, we would have 2^n actions that complete the initial state. Alternatively, some domains may lend themselves to achieve the same completion effect by applying a sequence of actions. In our example, each sequence of these actions generates one of the possible 2^n states.

An important limitation of both of the aforementioned approaches is that actions performed at the beginning of the plan make an explicit commitment to a *single* initial state. In many practical applications, such a compromise seems too excessive. Computationally, committing to a single state may lead the search astray. From a high-level perspective, committing to a single state produces assumptions that may be too restrictive, which may be undesirable. Both approaches outlined above have been used in the past to tackle diagnosis problems in which the initial state is unknown (Sohrabi, Baier, and McIlraith 2010; Haslum and Grastien 2011).

A Translation-Based Approach

In this section we propose an alternative translation of assumption-based planning into classical planning that builds on top of Palacios and Geffner's $K_{T,M}$ translation (2009) – henceforth denoted by P&G– which translates conformant planning into classical planning. The main objective of our translation is to avoid the excessive commitment exhibited by the naive translation of assumption-based planning into classical planning. We describe the basics of the translation, analyze its properties, and finally compare it to other extensions of P&G.

The $K^A_{T,M}$ Translation

Given a planning problem $P = (F, O, I, G)$, we generate a new planning problem $P' = (F', O', I', G')$; we call this process the $K^A_{T,M}$ translation, which builds on P&G $K_{T,M}$ translation. For each literal L we associate a set of *merges*, M_L. Each merge is a finite set of *tags*, which in turn are conjunctions of literals that are unknown in the initial state. Each merge characterizes a partition of the initial state in the sense that $I \models \bigvee_{t \in m} t$ is required to hold for each merge m.

A tag intuitively represents a partial completion of the initial state in which every $L \in t$ is initially true – it is a "case" in which L is initially true. Problem P' contains fluents of the form KL, for each $L \in F$, Kt and $K\neg t$ for each tag t, and KL/t for each $L \in F$ and each tag t in a merge of M_L. KL intuitively represents that L is *known*. KL/t represents the fact that L is known given that t is true in the initial state.

The main difference between P&G and our translation is that we consider a set of *assumption actions*, which allow the planner to assume that a tag t was true in the initial state. More specifically, given a set $\mathcal{T}$ of *assumable tags*, there is an assumption action associated to each $t \in \mathcal{T}$ that assumes t is true in the initial state. Instead of using the standard P&G merge actions, we use the contingent merge actions introduced by Albore, Palacios, and Geffner (2009). Furthermore, our translation augments P&G with additional conditional effects to handle assumptions. More precisely, P' is such that the following holds.

(1) $I' = \{KL \mid I \models L\} \cup \{KL/t \mid I, t \models L\} \cup \{ok\}$. I' differs from P&G's only in the ok fluent which is added to keep track of consistency and is explained in detail later.

(2) For each literal L, and each tag t in some merge of M_L, O contains the so-called *contingent merges* proposed originally by Albore, Palacios, and Geffner (2009); they are of the form $[\bigwedge_{t \in m}(KL/t \vee K\neg t)] \rightarrow KL$. These generalize P&G's merge actions for the case in which a tag t is refuted by assumptions.

(3) Like P&G, for each action a with conditional effect $C \rightarrow L$, O' contains the conditional effects $[\bigwedge_{c \in C} Kc/t] \rightarrow KL/t$ and $[\bigwedge_{c \in C} \neg K\neg c/t] \rightarrow \neg K\neg L/t$, for each tag t in some merge of M_L.

(4) In addition to P&G, for each tag t in the set of assumable tags, $\mathcal{T}$, we create an assumption action $Assume(t)$, with precondition $\neg K\neg t \wedge \neg Kt \wedge \neg K\neg t' \wedge \neg Kt'$ and effects Kt, $\neg K\neg t$, $K\neg t'$, $\neg Kt'$ for every tag t' that is inconsistent with t, i.e., contains the complement of a literal in t.

(5) For each merge set M_L that contains tag t, and each merge $m \in M_L$, the conditional effects $KL/t \rightarrow KL$, and $KL/t \wedge K\neg L \rightarrow \neg ok$ are added to the $Assume(t)$ action. The first conditional effect makes L known if it is the case that KL/t. The second conditional effect takes care of potential inconsistencies that could arise when assuming a literal that implies that L is known, when $\neg L$ is already known. In such cases the action deletes the fluent ok signaling inconsistency.

(6) For each action $a \in O$ the version of a in O' contains the precondition $ok \wedge \bigwedge_{L \in prec(a)} KL$.

(7) While building a plan, the planner should not make inconsistent assumptions. To illustrate this, consider that both $c_1 = \neg L_1 \vee L_2$ and $c_2 = \neg L_3 \vee L_1$ are clauses in I, and suppose a plan contains the action $Assume(L_3)$. Then action $Assume(\neg L_2)$ cannot consistently occur after $Assume(L_3)$ because that would imply that c_1 would be contradicted (L_1 is forced to be true by c_2 and the assumption of L_3). Thus whenever we assume a tag t, we may need to update the knowledge about other tags. We achieve this by adding specific conditional effects to assumption actions. Such effects reflect logical inferences among clauses defining the initial state and we obtain them by performing resolution. In our example, if one carries out a resolution step between c_1 and c_2, then we obtain the clause $c_3 = \neg L_3 \vee L_2$, from which it is straightforward that L_2 is forced to be true after assuming L_3. Using c_3 we write a new effect for $Assume(\neg L_3)$ that states KL_2 as a new effect. This idea can be extended further by computing *all* possible resolution steps with the clauses in the initial state.

In the general case, however, tags may be conjunctions of literals, and thus the relationship between different tags may not be entirely obvious by just looking at the clauses that result from resolution. In such a case, for each tag t, we add to I the clauses corresponding to $v_t \leftrightarrow [\bigwedge_{L \in t} L]$, where v_t is a new variable that represents a tag t. After carrying out all possible resolutions there will be clauses that only contain variables of the form v_t, and we only consider these clauses to generate the effects.

Let us denote by I^+ the set of clauses that result after performing resolution. Now we are ready to specify the conditional effects that are going to be added to $Assume(t)$. For each clause in $c \in I^+$ of the form $\{\ell_1, \ell_2, \ldots, \ell_n\}$, such that each ℓ_i is either t or $\neg t$ (or v_t or $\neg v_t$), for some tag t, we add the conditional effect $[\bigwedge_{\ell \in c \setminus \{\ell_i\}} K\neg\ell] \rightarrow K\ell_i$, for every $i \in \{1, \ldots, n\}$.

Note that the construction of I^+ is clearly worst-case exponential. Nevertheless, in practice, there are usually few resolution steps that can be made between clauses in I as usually I is formed by groups of clauses relatively independent of each other. Furthermore, when tags are of size 1, we do not need to add the clauses involving v_t as literals themselves represent their tags.

(8) Finally, $G' = \{KL \mid L \in G\} \cup \{ok\}$.

Just like P&G $K_{T,M}$ translation, our $K^A_{T,M}$ translation is *sound* in the following sense.

Theorem 4 *The $K^A_{T,M}$ translation is sound; i.e., if α is a plan for $K^A_{T,M}(P)$, then there is an assumption-based plan (ρ, α') for the original problem. Furthermore, (ρ, α') can be computed from α in linear time.*

The $K^A_i(P)$ Translation

As with P&G's $K_{T,M}$ translation, the $K^A_{T,M}$ translation does not define explicitly how the merges/tags are computed from the original problem. In addition, it provides no completeness guarantees. A practical realization of P&G's $K_{T,M}$ is given by the so-called K_i translation (Palacios and Geffner 2009). K_i defines an explicit way to compute merges. It is a sound translation (in the sense defined above). In addition, if i is not greater than the so-called *width* of the problem P, then it is also complete.

We have defined an analogous version of the K_i translation, that we call K^A_i. K^A_i is a version of K_i in which merges and tags are computed using the same procedure as for the case of K_i. Due to lack of space we cannot elaborate on this process, but we refer the reader to Palacios and Geffner's paper (2009) for reference. After the tags and merges are determined, however, it may be that the set of tags does not capture the set of assumable fluents. In such

a case, we create additional tags for those assumable fluents that are not captured.

Since K_i^A is a particular form of the translation $K_{T,M}$, we obtain that it is *sound* as a corollary of Theorem 4. Furthermore,

Theorem 5 *Given an assumption-based planning problem $P = (F, O, I, G, T)$, with width $w(P) \leq i$, the K_i^A translation is complete; i.e. if there exists an assumption-based plan (ρ, π) for P, in which ρ are conjunctions of literals in T, then a plan exists for $K_i^A(P)$.*

In the previous result, $w(P)$ is defined analogously to P&G.

Negative Results Given P, $K_i(P)$ is polynomial in the width of P (Palacios and Geffner 2009). Since our implementation involves a step in which previously we do a resolution fixpoint computation (Step (7)) we cannot guarantee that the K_i^A translation is polynomial on the width of P.

Preferred Assumption-Based Planning

The definition of an assumption-based plan allows the planner to assume any aspect of the state that can be constructed from the subset of assumable literals and consistently assumed. However, some assumptions will be more reasonable than others. E.g., in our car example, if it's summer, it may be much more reasonable to assume $batteryOk$ than that the car $hasGas$. In the cold of winter, the opposite may be true. To define the notion of a preferred assumption-based plan, we employ a preference relation $\preceq$, a transitive and reflexive relation in $\Pi \times \Pi$, where Π contains precisely all assumption-based plans for a particular planning instance (following Baier and McIlraith 2008). Plan optimality is defined in the obvious way given relation $\preceq$.

So far we have not discussed how to realize the preference relation $\prec$. The problem of specifying a preferred assumption-based plan is somewhat analogous to the task of specifying a preferred abductive explanations. (Indeed, Reiter and de Kleer established the relationship between abduction and assumption-based reasoning for nondynamical systems in (Reiter and de Kleer 1987).) There is a rich literature on this subject. Drawing on this analogy, we may define a syntactic, domain-independent notion of preference by restricting assumables to literals and preferring plans that assume the smallest number of literals. Appealing to domain-specific notions of preference, if probabilistic information is available, we may wish to define our preference relationship in terms of the likelihood of the assumable being true given what is known of the state. Alternatively, we could appeal to background knowledge described in default logic (e.g., (Brewka 1989)).

The problem with any of these approaches is that they provide a means of preferring assumables, but not necessarily a means of preferring assumption-based plans. In planning, the actions themselves also contribute to the quality of the plan – both the number of actions, and what actions are performed. A high-quality assumption-based plan may dictate making a number of reasonable assumptions at the outset, and then minimizing plan length, or it may be characterized by minimizing the number of assumptions and utilizing certain desirable actions.. As such a more compelling approach is to appeal to a planning formalism for preference-based planning such as PDDL3 (Gerevini et al. 2009), specifying assumptions just as we specify preferences, but with the variation that we penalize the use of assumptions (and actions) rather than the violation of preferences as is typically done in preference-based planning. We might alternatively characterize the problem as a net benefit problem (van den Briel et al. 2004). Detailed discussion of how best to specify such preferences is beyond the scope of this paper.

For the purposes of this paper, we will appeal to the uniform notion of action cost in order to characterize preferred assumption-based plans, rather than defining $\preceq$ directly. Specifically, given an assumption-based planning problem P, we build its translated instance $K_{T,M}^A(P) = (F, O, I, G)$, and then augment this instance to produce a *cost-based planning problem* $P_{\mathcal{C}} = (F, O, I, G)$ such that each action $a \in O$ has a non-negative cost $\mathcal{C}(a)$. Note that this means that actions of the form $Assume(t)$ will have a cost associated with them. Likewise, so do the domain actions and merge actions. The task reduces now to finding a cost-optimal plan.

Specifying how a domain expert would specify these preferences in the original problem specification and ensuring that the corresponding cost-based planning problem respects the induced $\prec$ relation can be achieved in a variety of ways. Detailed discussion of this issue is beyond the scope of this paper. For the purposes of illustrating some of the properties of this approach, we can directly and intuitively add costs to the translated classical planning problem, as we do in the section to follow.

Implementation and Experiments

The K_1^A translation was implemented in our A0 planner as an augmentation of Palacios and Geffner's T0 planner. In the absence of the specification of assumables, the assumables are set to all the fluents less those involved in G, precluding assumption-based plans that assume G. We use FF (Hoffmann and Nebel 2001) to generate classical plans with the translated domains and convert them back into assumption-based plans. To generate preferred assumption-based plans, we associate a cost with each action in the (translated) classical planning problem. The resulting cost-based planning problem is solved using Metric-FF and LAMA.

Since the notion of assumption-based planning is new, there are no systems to benchmark against. We sought instead to evaluate the running time of A0 + FF compared to an implementation of so-called *naive* assumption-based planning, and to various cost distributions for cost-based assumption-based planning. We also sought to assess gross properties of the translation: proportion of solution time; and size relative to its T0 counter part, and to the original problem.

Domains: We exploited four domains from the International Planning Competition (IPC) benchmark suite: *logistics*, *raokeys*, *coins*, and *blocks*. Experiments are still in progress for the latter two domains, but initial results are promising; details will be given in the full paper. In logistics, packages must be delivered to locations within

Table 1: Comparing the seven configurations with the total time to solve in seconds on the left and plan length on the right. The number of assumptions made appears in parentheses. The results are preliminary and experiments are in progress (X: unknown failure in back-end planner. TTO: time out during translation. STO: time out during solving classical plan). All experiments were run on a 2.80GHz machine with 2GB memory and a 30 minute timeout.

Prob	**Classical** (t(s)/len)		**Cost-Based Metric-FF** (t(s)/len)					**Cost-Based LAMA** (t(s)/len)				
	A0	naive	x=0.1	x=0.5	x=1	x=2	x=10	x=0.1	x=0.5	x=1	x=2	x=10
alog-1	0.06/44(2)	0.00/39(4)	0.26/39(2)	0.04/39(2)	0.03/39(2)	0.05/39(3)	0.29/39(3)	101.75/36(1)	37.58/36(1)	0.03/38(2)	0.02/38(2)	0.02/38(2)
alog-2	0.07/44(2)	0.01/ 51(16)	0.33/39(2)	0.04/39(2)	0.04/39(2)	0.06/39(3)	0.36/39(3)	6.03/36(1)	0.03/38(2)	0.02/38(2)	220.53/36(2)	251.66/36(2)
alog-3	0.06/44(2)	0.02/ 61(26)	0.33/39(2)	0.03/39(2)	0.04/39(2)	0.06/39(3)	0.35/39(3)	6.03/36(1)	0.03/38(2)	0.03/38(2)	0.03/38(2)	253.99/36(2)
alog-4	0.11/44(2)	0.27/ 129(94)	0.41/39(2)	0.05/39(2)	0.05/39(2)	0.06/39(3)	0.43/39(3)	6.35/36(1)	475.12/36(1)	0.04/41(2)	589.5/36(2)	0.05/41(2)
alog-5	0.06/50(2)	0.01/ 47(5)	0.39/45(2)	0.05/45(2)	0.05/45(2)	0.09/45(3)	0.69/45(3)	54.14/42(1)	38.67/44(2)	38.71/44(2)	38.53/44(2)	38.04/44(2)
alog-6	0.07/50(2)	0.01/ 50(8)	0.41/45(2)	0.06/45(2)	0.05/45(2)	0.11/45(4)	0.77/45(4)	4.45/42(1)	1.41/43(2)	1.49/43(2)	1.41/43(2)	1.45/43(2)
alog-7	0.07/52(3)	0.01/ 54(5)	0.03/51(3)	0.03/51(3)	0.03/51(3)	0.03/51(3)	0.07/51(3)	0.65/54(3)	0.67/54(3)	0.64/54(3)	0.66/54(3)	0.65/54(3)
alog-8	0.1/54(3)	0.01/ 50(6)	0.04/53(2)	0.03/53(2)	0.04/53(3)	0.05/52(4)	2.45/51(3)	13.6/55(2)	13.25/55(2)	13.32/55(2)	524.5/49(3)	669.47/51(3)
alog-9	0.08/56(5)	0.01/ 51(10)	0.10/61(2)	0.16/61(2)	0.15/61(4)	0.08/57(6)	9.80/51(4)	427.26/60(2)	423.53/52(3)	424.44/52(3)	423.99/52 93)	424.19/52(3)
alog-10	0.06/39(5)	0.01/ 36(6)	0.36/37(4)	0.04/39(5)	0.03/39(6)	0.06/36(5)	0.50/36(5)	8.93/33(1)	12.52/34(1)	0.03/37(4)	0.02/37(4)	156.15/33(4)
alog-11	0.14/55(6)	0.01/ 50(14)	1671.78/46(7)	0.21/48(9)	0.23/48(9)	1.56/52(11)	975.88/46(9)	53.21/43(1)	7.29/50(1)	71.76/51(6)	3.49/50(11)	99.78/49(11)
alog-12	0.16/61(8)	0.04/72(17)	160.64/61(5)	0.25/61(5)	0.32/58(4)	2.75/56(5)	STO	X	X	X	X	X
rao-2	0.05/10(2)	0.01/ 10(2)	0.02/ 10(2)	0.02/10(2)	0.03/ 10(2)	0.12/10(2)	33.33/10(2)	0.02/10(2)	0.02/10(2)	0.02/10(2)	0.02/10(2)	0.02/10(2)
rao-3	36.39/16(3)	0.04/ 24(14)	X	X	X	X	X	60.79/17(3)	57.93/18(3)	61.75/17(3)	75.57/17(4)	80.81/17(4)
rao-4	X	3.96/ 36	X	X	X	X	X	X	X	X	X	X
rao-5	X	X	X	X	X	X	X	X	X	X	X	X
coins-1	0.02/8(3)	0.01/10(4)	0.01/8(3)	0.01/8(3)	X	X	X	0.02/9(0)	X	0.02/8(3)	0.02/8(3)	0.02/8(3)
coins-7	0.05/13(5)	0.01/15(6)	0.03/13(5)	0.03/13(5)	0.04/13(5)	137.67/13(5)	STO	9.1/26(0)	0.12/13(4)	0.07/13(5)	0.07/13(5)	0.04/13(5)
coins-8	0.08/14(5)	0.01/16(6)	0.04/14(5)	0.03/14(5)	0.04/14(5)	137.86/14(5)	STO	10.07/26(0)	0.29/14(4)	0.07/14(5)	0.07/14(5)	0.04/14(5)
coins-17	0.61/21(7)	0.09/24(10)	X	X	X	X	X	137.61/22(6)	4.85/21(7)	3.71/21(7)	2.79/21(7)	2.93/21(7)
coins-29	X	31.51/70(33)	X	X	X	X	X	129.55/70(20)	149.45/68(20)	137.65/70(20)	132.58/69(20)	146.67/70(20)

and between cities. Within-city transportation is performed by truck, between-city transportation is performed by airplane. The domain specifies bi-directional routes that connect a subset of locations. The domain allows trucks to move between any two locations within a city. To convert this to an assumption-based planning problem, some routes were made uni-directional (corresponding to one-way streets, temporary closures, etc.) or closed completely. Trucks were provided with 3 levels of gas (and empty), each time a truck changes location, the gas level is decremented by one level. Certain locations are designated as having gas stations, permitting refueling. We refer to this modified domain as *alogistics*. The 12 instances we constructed varied in the number of cities and trucks (2-4), and locations within a city. Varying amounts of uncertainty were introduced into the initial state of each instance via unknown truck gas levels and locations, and the connectedness within cities.

The second domain we used was *raokeys*, a conformant planning benchmark from IPC-2008. The problem instance n has $n + 1$ keys and n lights each individually accessed through a different locked doors. Under each such light is a key. Starting with only $key0$ the agent must find all n keys. To get a particular key, the agent must have found the key that unlocks the corresponding door. It is unknown which key is at what light, and which door is unlocked by which key. Thus conformant plans are very long and must consider a combinatorially explosive number of possible initial states. In contrast, this problem is simple for a human: for the latest key found, try to find a key at each of the lights, repeat. We experimented with 4 instances of *raokeys*: raokeys2, raokeys3, raokeys4, raokeys5. These problem instances were left alone to see how the planner would address standard conformant planning problems.

Experiments: We ran 7 different experimental configurations on the 16 problem instances described above and 4 preliminary results from *coins*. **(1)** We ran A0 + FF on the translated domains. **(2)** We generated a naive assumption-based planning problem by augmenting each problem instance with actions that create each of the different consistent completions of the initial state, then solved with FF. **(3)–(7)** These configurations all relate to generating preferred assumption-based plans. The configurations differ with respect to the cost of the assumption actions relative to the domain actions. E.g., $x = 0.5$ denotes that assumption actions are twice as expensive as domain actions. All merge actions were assigned equal (low) cost. Instances solved via A0 + Metric-FF or LAMA.

Table 1 shows the results obtained on the seven configurations for 20 instances. On the classical settings, A0 does not take much more time than the naive method but makes far fewer assumptions. Problems alog-1 to alog-4, are all variants of the same problem instance but with progressively more unimportant uncertainty. This domain is contrasted by the *raokeys* problem instances in which A0 seems to take exponentially more time while the naive method works fairly well. Strong conclusions cannot be made however because of the small number of instances that can be solved. This is conjecture, as there are only two and three data points to draw from, however it fits with the explanation of runaway dependency provided earlier. The reason the naive method is able to handle the small instances (raokeys2, raokeys3, and raokeys4) is because the conditional dependencies are not considered, it merely needs to set the unknown fluents and find a plan from there. This method *is* overcome once there

are 5 lights, however, as the number of fluents that need to be set is large enough to overwhelm it.

The cost-based run on metric-FF showed a clear trend that $x = 0.1$ and $x = 10$ took longer to solve than values of x closer to 1. All the *alogistic* instances require both assumptions and regular actions to be solved. When the cost discrepancies are high, the search may be too focused building plan prefixes with the cheapest action, without considering other (useful) actions.

This is contrasted by the trend in raokeys2 where it takes longer to solve the less assumptions are penalized. Considering that this is a fully conformant problem instance it is reasonable that favouring assumption actions (which aren't necessary) can make it harder to find (prove) the optimal plan.

Finally, the number of assumptions does indeed decrease as they are penalized further. In these instances it isn't possible to have a large difference in the number of assumptions because on the one hand a few assumptions are *required* for each instance, and on the other hand only a few assumptions are possible to make in one plan without inconsistencies With further testing on problem instances that either allow or require many more assumptions we believe the trend will be more apparent and interesting. It would be especially good to test on some domains that can be solved fully conformantly *and* have many assumptions that are useful and not mutex.

We evaluated the size of our A0 translations relative to the original problem and to a comparable T0 translation. Given the diversity of domains and the small number, our observations are somewhat anecdotal. For the *alogistics* domains, A0 and T0 performed reasonably consistently. The number of atoms was approximately double that in the original domain and there was a very small increase in the number of actions. In the *raokeys* domains A0 and T0 both doubled the number of actions, but there was no other clear trend, except that A0 had one or two orders of magnitude more conditional effects than the original domain, whereas T0 only had a constant factor increase. These large number of conditional effects are due to the dependencies between clauses in the domain which results in a high number of clauses that can be produced by the resolution algorithm.

We also evaluated the proportion of solution time dedicated to the translation. For the *alogistics* domains this ranged from 5 - 25%, whereas with the *raokeys* domain, it was closer to 50%. On the other hand, A0 took about $0.1 - 0.6$ times the amount of time to translate *alogistics* problem instances as T0. On raokeys2 the time spent was about the same, but on raokeys3 A0 took almost 100 times as long. Again, this is due to the unit propagation required for the assumption actions.

Summary and Concluding Remarks

In this paper we introduce the notion of assumption-based planning. We provide a formal characterization of assumption-based planning, establishing a correspondence to conformant planning. Exploiting this correspondence, we provide a translation of an assumption-based planning problem to a classical planning problem, building on the popular translation developed by P&G. We prove the soundness and completeness of our translation. This provides us with a means of generating assumption-based plans using classical planners. We also argue for the merit of preferred assumption-based plans and propose a means of realizing such plans using cost-based planning. We describe A0, a planner that addresses the subset of initial state assumption-based planning problems and present experiments that illustrate the viability of our approach and that assess some properties of our translation.

While this paper explores the generation of assumption-based plans via a translation to classical planning, the correspondence to conformant planning opens the door to adapting a variety of conformant planners for assumption-based planning (e.g., (To, Son, and Pontelli 2010)). Beyond planning, the assumption-based planning paradigm has compelling applications in diagnosis and verification of dynamical systems.

Acknowledgements: We gratefully acknowledge funding from the Natural Sciences and Engineering Research Council of Canada (NSERC). Jorge Baier was partly funded by Fondecyt grant no. 11110321.

References

Albore, A., and Bertoli, P. 2004. Generating safe assumption-based plans for partially observable, nondeterministic domains. In *Proc. of the 19th National Conference on Artificial Intelligence (AAAI)*, 495–500.

Albore, A., and Bertoli, P. 2006. Safe ltl assumption-based planning. In *Proc. of the 16th International Conference on Automated Planning and Scheduling (ICAPS)*, 193–202.

Albore, A., and Geffner, H. 2009. Acting in partially observable environments when achievement of the goal cannot be guaranteed. In *Proc. of ICAPS Workshop on Planning and Plan Execution for Real-World Systems*.

Albore, A.; Palacios, H.; and Geffner, H. 2009. A translation-based approach to contingent planning. In *Proc. of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, 1623–1628.

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence* 116(1-2):123–191.

Baier, J. A., and McIlraith, S. A. 2008. Planning with preferences. *Artificial Intelligence Magazine* 29(4):25–36.

Bonet, B., and Geffner, H. 2011. Planning under partial observability by classical replanning: Theory and experiments. In *Proc. of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*.

Brewka, G. 1989. Preferred subtheories: An extended logical framework for default reasoning. In *Proc. of the 11th International Joint Conference on Artificial Intelligence (IJCAI)*, 1043–1048.

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69(1-2):165–204.

Conrad, P. R., and Williams, B. C. 2011. Drake: An efficient executive for temporal plans with choice. *Journal of Artificial Intelligence Research* 42:607–659.

Console, L.; Dupre, D. T.; and Torasso, P. 1989. Abductive reasoning through direct deduction from completed domain models. In Ras, Z., ed., *Methodologies for Intelligent Systems 4*. North Holland. 175 – 182.

de Giacomo, G., and Vardi, M. Y. 1999. Automata-theoretic approach to planning for temporally extended goals. In Biundo, S., and Fox, M., eds., *ECP*, volume 1809 of *LNCS*, 226–238. Durham, UK: Springer.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence* 173(5-6):619–668.

Göbelbecker, M.; Gretton, C.; and Dearden, R. 2011. A switching planner for combined task and observation planning. In *Proc. of the 26th AAAI Conference on Artificial Intelligence (AAAI)*.

Haslum, P., and Grastien, A. 2011. Diagnosis as planning: Two case studies. In *Proc. of the International Scheduling and Planning Applications workshop (SPARK)*.

Haslum, P., and Jonsson, P. 1999. Some results on the complexity of planning with incomplete information. In *Proc. of the 5th European Conference on Planning (ECP)*, 308–318.

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

Lin, F., and Reiter, R. 1997. How to progress a database. *Artif. Intell.* 92(1-2):131–167.

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

Pellier, D., and Fiorino, H. 2004. Assumption-based planning. In *In Proceedings of the International Conference on Advances in Intelligence Systems Theory and Applications, Luxemburg*.

Pellier, D., and Fiorino, H. 2005. Multi-agent assumption-based planning. In *Proc. of the 19th International Joint Conference on Artificial Intelligence (IJCAI)*, 1717–1718.

Poole, D.; Goebel, R.; and Aleliunas, R. 1987. Theorist: a logical reasoning system for defaults and diagnosis. In Cercone, N., and McCalla, G., eds., *The Knowledge Frontier: Essays in the Representation of Knowledge*. New York: Springer Verlag. 331–352.

Reiter, R., and de Kleer, J. 1987. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proc. of the 6th National Conference on Artificial Intelligence (AAAI)*, 183–189.

Sohrabi, S.; Baier, J.; and McIlraith, S. A. 2010. Diagnosis as planning revisited. In *Proc. of the 12th International Conference on Knowledge Representation and Reasoning (KR)*, 26–36.

Sohrabi, S.; Baier, J. A.; and McIlraith, S. A. 2011. Preferred explanations: Theory and generation via planning. In *Proc. of the 26th AAAI Conference on Artificial Intelligence (AAAI)*, 261–267.

To, S. T.; Son, T. C.; and Pontelli, E. 2010. A new approach to conformant planning using cnf*. In *Proc. of the 20th International Conference on Automated Planning and Scheduling (ICAPS)*, 169–176.

van den Briel, M.; Nigenda, R. S.; Do, M. B.; and Kambhampati, S. 2004. Effective approaches for partial satisfaction (over-subscription) planning. In *Proc. of the 19th National Conference on Artificial Intelligence (AAAI)*, 562–569.

Waldinger, R. 1977. Achieving several goals simultaneously. In *Machine Intelligence 8*. Edinburgh, Scotland: Ellis Horwood. 94–136.

Stochastic Shortest Path MDPs with Dead Ends

Andrey Kolobov Mausam Daniel S. Weld
{akolobov, mausam, weld}@cs.washington.edu
Dept of Computer Science and Engineering
University of Washington
Seattle, USA, WA-98195

Abstract

Stochastic Shortest Path (SSP) MDPs is a problem class widely studied in AI, especially for probabilistic planning. To make value functions bounded, SSPs make the severe assumption of no dead-end states. Thus, they are unable to model various scenarios that may have catastrophic events (e.g., sending a rover on Mars). Even though MDP algorithms are used for solving problems with dead-ends, a principled theory of SSP extensions that would allow dead ends, including theoretically sound algorithms for solving them, has been lacking. In this paper, we propose three new MDP classes that admit dead ends with increasingly weaker assumptions. We present Value Iteration-based as well as the more efficient heuristic search algorithms for optimally solving each class, and explore theoretical relationships between these classes. We also conduct a preliminary empirical study comparing the performance of our algorithms on different MDP classes, especially on scenarios with unavoidable dead ends.

Introduction

Stochastic Shortest Path (SSP) MDPs (Bertsekas 1995) is a class of probabilistic planning problems thoroughly studied in AI. They describe a wide range of scenarios where the objective of the agent is to reach a goal state in the least costly way in expectation from any non-goal state using actions with probabilistic outcomes.

While SSPs are a popular model, they have a serious limitation. They assume that a given MDP has at least one *complete proper policy*, a policy that reaches the goal from any state with 100%-probability. Basic algorithms for solving SSP MDPs, such as Value Iteration (VI) (Bellman 1957), fail to converge if this assumption does not hold. In the meantime, this requirement effectively disallows the existence of *dead ends*, states from which reaching the goal is impossible, and of catastrophic events that lead to these states. Such catastrophic failures are a possiblity to be reckoned with in many real-world planning problems, be it sending a rover on Mars or navigating a robot in a building with staircases. Thus, insisting on the absence of dead ends significantly limits the applicability of SSPs. Moreover, verifying that a given MDP has no dead ends can be nontrivial, further complicating the use of this model.

Researchers have realized that allowing dead ends in goal-oriented MDPs would break some existing methods for solving them (Little and Thiebaux 2007). They have also suggested algorithms that are aware of the possible presence of dead-end states (Kolobov, Mausam, and Weld 2010) and try to avoid them when computing a policy (Keyder and Geffner 2008; Bonet and Geffner 2005). However, these attempts have lacked a theoretical analysis of how to incorporate dead ends into SSPs in a principled way, and what the optimization criteria in the presence of dead ends should be. This paper bridges the gap by introducing three new MDP classes with progressively weaker assumptions about the existence of dead ends, analyzing their properties, and presenting optimal VI-like and more efficient heuristic search algorithms for them.

The first class we present, SSPADE, is a small extension of SSP that has well-defined easily-computable optimal solutions if the dead ends are present but are avoidable *provided that the process starts at a known initial state* s_0.

The second and third classes introduced in this paper admit that dead ends may exist and the probability of running into them from the initial state may be positive no matter how hard the agent tries. If the chance of a catastrophic event under any policy is nonzero, a key question is: should we prefer policies that minimize the expected cost of getting to the goal even at the expense of an increased risk of failure, or those that reduce the risk of failure above all else?

The former criterion characterizes scenarios where entering a dead end, while highly undesirable, has a finite "price". For instance, suppose the agent buys an expensive ticket for a concert of a favorite band in another city, but remembers about it only on the day of the event. Getting to the concert venue requires a flight, either by hiring a business jet or by a regular airline with a layover. The first option is very expensive but almost guarantees making the concert on time. The second is much cheaper but, since the concert is so soon, missing the connection, a somewhat probable outcome, means missing the concert. Nonetheless, the cost of missing the concert is only the price of the ticket, so a rational agent would probably choose to travel with a regular airline. Accordingly, one of the MDP classes we propose, fSSPUDE, assumes that the agent can put a price (penalty) on ending up in a dead end state and wants to compute a policy with the least expected cost (including the possible penalty). While seemingly straightforward, this intuition is tricky to operationalize because of several subtleties, e.g., MDP description does not specify *which* states are dead ends as it does for goals. This paper overcomes these subtleties and shows how fSSPUDE can be solved with easy modifi-

cations to SSP algorithms.

On the other hand, consider the task of planning an ascent to the top of Mount Everest for a group of human alpinists. To any human, the price of their own life can be taken as infinite; therefore, for such an undertaking a natural primary objective is to maximize the *probability* of getting to the goal alive (i.e. minimizing the chance of getting into an accident, a dead-end state). However, of all policies that maximize this chance, we would prefer the least costly one (in expectation). This is the optimization objective of the third MDP class described in this paper, iSSPUDE. Solving this MDP type is much more involved than handling the previous two, and we introduce two novel algorithms for it.

Intuitively, the objectives of fSSPUDE and iSSPUDE MDPs are related — as the fSSPUDE dead-end penalty gets bigger, the optimal policies of the two classes coincide. We provide a theoretical and an empirical analysis of this insight, showing that solving fSSPUDE may indeed yield an optimal policy for iSSPUDE if the dead-end penalty is high enough.

Thus, the paper makes four contributions: (1) three new goal-oriented MDP models that admit the existence of dead-end states; (2) optimal VI and heuristic search algorithms for solving them; (3) theoretical results describing equivalences among problems in these classes; and (4) an empirical evaluation tentatively answering the question: which class should be used when modeling a given scenario involving unavoidable dead ends?

Background and Preliminaries

SSP MDPs. In this paper, we extend an MDP class known as the Stochastic Shortest Path (SSP) problems with an optional initial state, defined as tuples of the form $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $\mathcal{T}$ is a transition function $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ that gives the probability of moving from s_i to s_j by executing a, $\mathcal{C}$ is a map $\mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that specifies action costs, $\mathcal{G}$ is a set of (absorbing) goal states, and s_0 is an optional start state. For each $g \in G$, $\mathcal{T}(g, a, g) = 1$ and $\mathcal{C}(g, a) = 0$ for all $a \in A$, which forces the agent to stay in g forever while accumulating no reward.

An SSP must also satisfy the following conditions: (1) Each $s \in \mathcal{S}$ must have at least one *complete proper policy*, i.e. a rule prescribing an action to take for any given state with which an agent can reach a goal state from any state with probability 1. (2) Every *improper* policy must incur the cost of ∞ from all states from which it cannot reach the goal with probability 1.

When the initial state is unknown, solving an SSP MDP means finding a policy whose execution from any state allows an agent to reach a goal state while incurring the least expected cost. We call such a policy *complete optimal*, and denote any complete policy as π. When the initial state is given, we are interested in computing an *optimal (partial) policy rooted at* s_0, i.e. one that reaches the goal in the least costly way from s_0 and is defined for every state it can reach from s_0 (though not necessarily for other states).

To make the notion of policy cost more concrete, we define a *cost function* of as a mapping $J : \mathcal{S} \to \mathbb{R} \cup \{\infty\}$ and let random variables S_t and A_t denote respectively the state of the process after t time steps and A_t the action selected in S_t. Then, the cost function J^π of policy π is

$$J^\pi(s) = \mathbb{E}^\pi_s \left[\sum_{t=0}^{\infty} \mathcal{C}(S_t, A_t) \right] \quad (1)$$

In other words, the cost of policy π at a state s is the expectation of the total cost the policy incurs if the execution of π is started in s. In turn, every cost function J has a policy π^J that is *J-greedy*, i.e. that satisfies

$$\pi^J(s) = \arg\min_{a \in \mathcal{A}} \left[\mathcal{C}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') J(s') \right] \quad (2)$$

Optimally solving an SSP MDP means finding a policy that minimizes J^π. Such policies are denoted π^*, and their cost function $J^* = J^{\pi^*}$, called the *optimal cost function*, is defined as $J^* = \min_\pi J^\pi$. J^* also satisfies the following condition, the *Bellman equation*, for all $s \in \mathcal{S}$:

$$J(s) = \min_{a \in \mathcal{A}} \left[\mathcal{C}(s, a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s, a, s') J(s') \right] \quad (3)$$

Value Iteration for SSP MDPs. The Bellman equation suggests a dynamic programing method of solving SSPs, known as Value Iteration (VI_{SSP}) (Bellman 1957). VI_{SSP} starts by initializing state costs with an arbitrary *heuristic cost function* $\hat{J}$. Afterwards, it executes several sweeps of the state space and updates every state during every sweep by using the Bellman equation (3) as an assignment operator, the *Bellman backup operator*. Denoting the cost function after the i-th sweep as J_i, it can be shown that the sequence $\{J_i\}_{i=1}^{\infty}$ converges to J^*. A complete optimal policy π^* can be derived from J^* via Equation 2 .

Heuristic Search for SSP MDPs. Because it stores and updates the cost function for the entire $\mathcal{S}$, VI_{SSP} can be slow and memory-inefficient even on relatively small SSPs. However, if the initial state s_0 is given we are interested in computing $\pi^*_{s_0}$, an optimal policy from s_0 only, which typically does not visit (and hence does not need to be defined for) all states. This can be done with a family of algorithms based on VI called *heuristic search*. Like VI, these algorithms need to be initialized with a heuristic $\hat{J}$. However, if $\hat{J}$ is *admissible*, i.e. satisfies $\hat{J}(s) \leq J^*(s)$ for all states, then heuristic search algorithms can often compute J^* for the states relevant to reaching the goal from s_0 without updating or even memoizing costs for many of the other states. At an abstract level, the operation of any heuristic search algorithm is represented by the FIND-AND-REVISE framework (Bonet and Geffner 2003a). As formalized by FIND-AND-REVISE, any heuristic search algorithm starts with an admissible $\hat{J}$ and explicitly or implicitly maintains the graph of a policy greedy w.r.t. the current J, updating the costs of states *only in this graph* via Bellman backups. Since the initial $\hat{J}$ makes many states look "bad" a-priori, they never end up in the greedy graph and hence never have to be stored or updated. This makes heuristic search algorithms, e.g. LRTDP (Bonet and Geffner 2003b), work more efficiently than VI and still produce an optimal $\pi^*_{s_0}$.

GSSP and MAXPROB MDPs. Unfortunately, many interesting probabilistic planning scenarios fall outside of the SSP MDP class. One example is MAXPROB MDPs (Kolobov et al. 2011), goal-oriented problems where the objective is to maximize the probability of getting to the goal, not minimize the cost. Namely, consider the following definition:

Definition For an MDP with a set of goal states $\mathcal{G} \subset \mathcal{S}$, the *goal-probability function* of a policy π, denoted P^π, gives the probability of reaching the goal from any state s. Mathematically, letting $S_t^{\pi_s}$ be a random variable denoting a state the MDP may end up if policy π is executed starting in state s for t time steps,

$$P^\pi(s) = \sum_{t=1}^{\infty} P[S_t^{\pi_s} = g \in \mathcal{G}, S_{t'}^{\pi_s} = s \notin \mathcal{G} \; \forall \, 1 \leq t' < t] \quad (4)$$

Each term in the above summation denotes the probability that, if π is executed starting at s, the MDP ends up in a goal state at step t and not earlier. Since once the system enters a goal state it stays in that goal state forever, the sum of all such terms is the probability of the system ever entering a goal state under π.

Solving a MAXPROB means finding the optimal goal-probability function, one that satisfies $P^*(s) = \arg\max_\pi P^\pi(s)$ for all states. Alternatively $1 - P^*(s)$ can be interpreted as the smallest probability of running into a dead end from s for any policy. Thus, solving a MAXPROB derived from a goal-oriented MDP by discarding action costs can be viewed as a way to identify *dead ends*:

Definition For a goal-oriented MDP, a *dead-end state* (or dead end, for short) is a state s for which $P^*(s) = 0$.

MAXPROBs, SSPs themselves, and many other MDPs belong to the broader class of Generalized SSP MDPs (GSSPs) (Kolobov et al. 2011). GSSPs are defined as tuples $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$ of the same form as SSPs, but relax both of the additional conditions in the SSP definition. In particular, they do not require the existence of a complete proper policy as SSPs do. Without going further into the GSSP definition specifics, we note that algorithms for solving GSSPs, discussed below, will help us in designing solvers for the MDP classes introduced in this paper.

Value Iteration for GSSP MDPs. In the case of SSPs, VI_{SSP} yields a complete optimal policy for these MDPs independently of the initializing heuristic $\hat{J}$. For a GSSP MDP, such a policy need not exist, so neither does an analog of VI_{SSP} that works for all problems in this class. However, for MAXPROB, a subclass of GSSP particularly important to us in this paper, such an algorithm, called VI_{MP}, can be designed. Like VI_{SSP}, VI_{MP} can be initialized with an arbitrary heuristic function, but instead of the Bellman backup operator it uses its generalized version that we call *Bellman backup with Escaping Traps* (BET) in this paper. BET works by first updating the initial heuristic function with Bellman backup, until it arrives at a fixed-point function $P^\times$. For SSPs, Bellman backup has only one fixed point, the optimal P^*, so we would stop here. However, for GSSPs (and MAXPROB in particular) this is not the case — P^* is only *one of* Bellman backup's fixed points, and the current fixed point $P^\times$ may not be equal to P^*. Crucially, to check whether $P^\times$ is optimal, BET applies the *trap elimination* operator to it, which involves constructing the transition graph that uses actions of *all* policies greedy w.r.t. $P^\times$. If $P^\times \neq P^*$, trap elimination generates a new, non-fixed-point $P^{\times'}$, on which BET again acts with Bellman backup, and so on. The fact that VI_{MP} and FRET, the heuristic search framework for GSSPs considered below, sometimes need to build a greedy transition graph w.r.t. a cost function is important for analyzing the performance of algorithms introduced in this paper.

VI_{MP}'s main property, whose proof is a straightforward extension of the results in the GSSP paper (Kolobov et al. 2011), is similar to VI_{SSP}'s:

Theorem 1. *On MAXPROB MDPs, VI_{MP} converges to the optimal goal-probability function P^* independently of the initializing heuristic function $\hat{J}$.*

Heuristic Search for GSSP MDPs. Although a complete optimal policy does not necessarily exist for a GSSP, one rooted at s_0 always does and can be found by any heuristic search algorithm conforming to an FIND-AND-REVISE analogue for GSSPs, FRET (Kolobov et al. 2011). Like FIND-AND-REVISE, FRET guarantees convergence to $\pi^*_{s_0}$ if the initializing heuristic is admissible.

MDPs with Avoidable Dead Ends

All definitions of the SSP class in the literature (Bertsekas 1995; Bonet and Geffner 2003a; Kolobov et al. 2011) require that the goal be reachable with 100%-probability from every state in the state space, even when initial state s_0 is known and the objective is to find an optimal policy rooted only at that state. We first extend SSPs to the easiest case — when dead ends exist but can be avoided entirely from s_0.

Definition A Stochastic Shortest Path MDP with Avoidable Dead Ends (SSPADE) is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, s_0 \rangle$ where $\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}$, and s_0 are as in the SSP MDP definition, under the following conditions:

- The initial state s_0 is known.
- There exists at least one proper policy rooted at s_0.
- Every *improper* policy must incur the cost of ∞ from at least one state reachable by it from s_0.

Solving a SSPADE MDP means finding a policy $\pi^*_{s_0}$ rooted at s_0 that satisfies $\pi^*(s_0) = \arg\min_\pi J^\pi(s_0)$.

Value Iteration: Even though dead ends may be avoided with optimal policy from s_0, they are still present in the state space. Thus VI_{SSP}, which operates on the entire state space, still does not converge – the optimal costs for dead ends are infinite. One might think that we may be able to adapt VI_{SSP} by restricting computation to the subset of states reachable by s_0. However, even this is not true, since SSPADE requirements do not preclude dead-ends reachable from s_0. Overall, for VI to work we need to detect divergence of state cost sequences – an unsolved problem, to our knowledge.

Heuristic Search: Although VI does not terminate for SSPADE, heuristic search algorithms do. This is because:

Theorem 2. $SSPADE \subset GSSP$.

Proof sketch. SSPADE directly satisfies all requirements of the GSSP definition (Kolobov et al. 2011). □

In fact, we can also show that heuristic search for SSPADE only needs the regular Bellman backup operator (instead of the BET operator). That is, all FIND-AND-REVISE framework heuristic search algorithms such as LRTDP and LAO* work without modifications for this class.

Intuitively, FIND-AND-REVISE starts with admissible (lower bound to optimal) state costs. As FIND-AND-REVISE updates them, costs of dead ends grow without bound, while costs of other states converge to a finite value. Thus, dead ends become unattractive and drop out of the greedy policy graph rooted at s_0.

MDPs with Unavoidable Dead Ends

In this section, our objective is threefold: (1) to motivate the semantics of SSP extensions that admit unavoidable dead ends; (2) to state intuitive policy evaluation criteria and thereby induce the notion of optimal policy for models in which the agent pays finite and infinite penalty for visiting dead ends; (3) to formally define MDP class SSPUDE with subclasses fSSPUDE and iSSPUDE that model such finite- and infinite-penalty scenarios.

Consider an *improper* SSP MDP, one that conforms to the SSP definition except for the requirement of proper policy existence. In such an MDP, the objective of finding a policy that minimizes the expected cost of reaching the goal becomes ill-defined. It implicitly assumes that for at least one policy, the cost incurred by all of the policy's trajectories is finite, which is true only for proper policies, whose every trajectory terminates at the goal. Thus, all policies in an improper SSP may have an infinite expected cost, making the cost criterion unhelpful for selecting the "best" policy.

We suggest two ways of amending the optimization criterion to account for unavoidable dead ends. The first is to assign a finite positive penalty D for visiting a dead end. The semantics of an improper SSP altered this way would be that the agent pays D when encountering a dead end, and the process stops. However, this straightforward modification to the MDP cannot be directly operationalized, since the set of dead-ends is not known a-priori and need to be inferred while planning. Moreover, this definition also has a caveat – it may cause non-dead-end states that lie on potential paths to a dead end to have higher costs than dead ends themselves. For instance, imagine a state s whose only action leads with probability $(1-\epsilon)$ to a dead end, with probability $\epsilon > 0$ to the goal, and costs $\epsilon(D+1)$. A simple calculation shows that $J^*(s) = D + \epsilon > D$, even though reaching the goal from s *is* possible. Moreover, notice that this semantic paradox cannot be resolved just by increasing the penalty D.

Therefore, we change the semantics of the finite-penalty model as follows. Whenever the agent reaches *any* state with the expected cost of reaching the goal equaling D or greater, the agent simply pays the penalty D and "gives up", i.e. the process stops. Intuitively, this setting describes scenarios where the agent can put a price on how desirable reaching the goal is. For instance, in the example from the introduction involving a concert in another city, paying the penalty corresponds to deciding not to go to the concert, i.e. foregoing the pleasure it would have derived from attending the performance.

The benefit of putting a "cap" on any state's cost as described above is that the cost of a state under any policy becomes finite, formally defined as

$$\mathrm{J}^{\mathrm{F}\pi}(s) = \min\left\{D, \mathbb{E}\left[\sum_{t=0}^{\infty} \mathcal{C}(S_t^{\pi_s}, A_t^{\pi_s})\right]\right\} \tag{5}$$

It can be shown that for an improper SSP, there exists an optimal policy π^* [1], one that satisfies

$$\pi^*(s) = \arg\min_{\pi} \mathrm{J}^{\mathrm{F}\pi}(s) \;\forall\, s \in \mathcal{S} \tag{6}$$

As we show shortly, we can find such a policy using the expected-cost analysis similar to that for ordinary SSP MDPs. The intuitions just described motivate the fSSPUDE MDP class, defined at the end of this section.

The second way of dealing with dead ends we consider in this paper is to view them as truly irrecoverable situations and assign $D = \infty$ for visiting them. As a motivation, recall the example of planning a climb to the top of Mount Everest. Since dead ends here cannot be avoided with certainty and the penalty of visiting them is ∞, comparing policies based on the expected cost of reaching the goal breaks down — they all have an infinite expected cost. Instead, we would like to find a policy that maximizes the probability of reaching the goal and whose expected cost *over the trajectories that reach the goal* is the smallest.

To describe this policy evaluation criterion more precisely, let $S_t^{\pi_s+}$ be a random variable denoting a distribution over states s' for which $P^\pi(s') > 0$ and in which the MDP may end up if policy π is executed starting from state s for t steps. Put differently, $S_t^{\pi_s+}$ is just a restriction of the variable $S_t^{\pi_s}$ used previously to states from which policy π can reach the goal. Using the $S_t^{\pi_s+}$ variables, we can mathematically evaluate π with two ordered criteria by defining the cost of a state as an ordered pair

$$\mathrm{J}^{\mathrm{I}\pi}(s) = (P^\pi(s), [J^\pi|P^\pi](s)) \tag{7}$$

$$\text{where} \quad [\mathrm{J}^\pi|\mathrm{P}^\pi](\mathrm{s}) = \mathbb{E}\left[\sum_{\mathrm{t}=0}^{\infty} \mathcal{C}(\mathrm{S}_\mathrm{t}^{\pi_\mathrm{s}+}, \mathrm{A}_\mathrm{t}^{\pi_\mathrm{s}})\right] \tag{8}$$

Specifically, we write $\pi(s) \prec \pi'(s)$, meaning π' is preferable to π at s, whenever $\mathrm{J}^{\mathrm{I}\pi}(s) \prec \mathrm{J}^{\mathrm{I}\pi'}(s)$, i.e. either $P^\pi(s) < P^{\pi'}(s)$ or $P^\pi(s) = P^{\pi'}(s)$ and $[J^\pi|P^\pi](s) > [J^{\pi'}|P^{\pi'}](s)$. Notice that the second criterion is used conditionally, only if two policies are equal in terms of the probability of reaching the goal, since maximizing this probability is the foremost priority. Note also that if $P^\pi(s) =$

[1] We implicitly assume that one of the optimal policies is *deterministic Markovian* — a detail we can actually prove but choose to gloss over in this paper for clarity.

$P^{\pi'}(s) = 0$ then both $[J^{\pi}|P^{\pi}](s)$ and $[J^{\pi'}|P^{\pi'}](s)$ are ill-defined. However, since neither π nor π' can reach the goal from s, we define $[J^{\pi}|P^{\pi}](s) = [J^{\pi'}|P^{\pi'}](s) = 0$ for such cases, and hence $\text{JI}^{\pi}(s) = \text{JI}^{\pi'}(s)$.

As in the finite-penalty case, we can demonstrate that there exists a policy π^* that is at least as large as all others at all states under the $\prec$-ordering above and hence is optimal, i.e.

$$\pi^*(s) = \arg\max_{\prec\pi} \text{JI}^{\pi}(s) \; \forall \; s \in \mathcal{S} \tag{9}$$

We are now ready to capture the above intuitions in a definition of the SSPUDE MDP class and its subclasses fSSPUDE and iSSPUDE:

Definition An SSP with Unavoidable Dead Ends (SSPUDE) MDP is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, D, s_0 \rangle$, where $\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}$, and s_0 are as in the SSP MDP definition, $D \in \mathbb{R}^+ \cup \{\infty\}$ is a penalty incurred if a dead end state is visited, and for which every improper policy incurs an infinite expected cost, as defined by Equation 1, at all states from which it cannot reach the goal with probability 1.

If $D < \infty$, the MDP is called an fSSPUDE MDP, and its optimal solution is a policy π^* satisfying $\pi^*(s) = \min_{\pi} \text{JF}^{\pi}(s)$ for all $s \in \mathcal{S}$.

If $D = \infty$, the MDP is called an iSSPUDE MDP, and its optimal solution is a policy π^* satisfying $\pi^*(s) = \max_{\prec\pi} \text{JI}^{\pi}(s)$ for all $s \in \mathcal{S}$.

Note the clause requiring that in all SSPUDE MDPs every improper policy must incur an infinite cost if evaluated with Equation 1. This clause excludes, for instance, MDPs with zero-cost loops, in which the agent can stay forever without reaching the goal while accumulating only a finite amount of penalty.

Our iSSPUDE class is related to multi-objective MDPs, which model problems with several competing objectives, e.g., total time, monetary cost, etc. (Chatterjee, Majumdar, and Henzinger 2006; Wakuta 1995). They jointly optimize these metrics and return a pareto-set of all non-dominated policies. Unfortunately, such solutions are impractical due to their higher computational requirements. Moreover, the objective function of probability of goal achievement converts the problem into a GSSP and hence cannot be easily included in those models.

The Case of a Finite Penalty

Equation 5 tell us that for an fSSPUDE instance, the cost of any policy at any state is finite. Intuitively, this implies that fSSPUDE should be no harder to solve than SSP. This intuition is confirmed by this following result:

Theorem 3. *fSSPUDE = SSP.*

Proof sketch. To show that every fSSPUDE MDP $M_{fSSPUDE}$ can be converted to an SSP MDP, we augment the action set $\mathcal{A}$ of fSSPUDE with a special action a' that causes a transition to a goal state with probability 1 and that costs D. This MDP is an SSP, since reaching the goal with certainty is possible from every state. At the same time, the optimization criteria of fSSPUDE and SSP clearly yield the same set of optimal policies for it.

To demonstrate that every SSP MDP M_{SSP} is also an fSSPUDE MDP, for every M_{SSP} we can construct an equivalent fSSPUDE MDP by setting $D = J^*(s)$. The set of optimal policies of both MDPs will be the same. (Note, however, that the conversion procedure is impractical since it assumes that we know $J^*(s)$ before solving the MDP.) □

The above conversion from fSSPUDE to SSP immediately suggests solving fSSPUDE with modified versions of standard SSP algorithms, as we describe next.

Value Iteration: Theorem 3 implies that JF^*, the optimal cost function of an fSSPUDE MDP, must satisfy the following modified Bellman equation:

$$J(s) = \min \left\{ D, \min_{a \in \mathcal{A}} \left[\mathcal{C}(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{T}(s,a,s') J(s') \right] \right\} \tag{10}$$

Moreover, it tell us that π^* of an fSSPUDE must be greedy w.r.t to JF^*. Thus, an fSSPUDE can be solved with arbitrarily initialized VI_{SSP} that uses Equation 10 for updates.

Heuristic Search: By the same logic as above, all FIND-AND-REVISE algorithms and their guarantees apply to fSSPUDE MDPs if they use Equation 10 in lieu of Bellman backup. Thus, all heuristic search algorithms for SSP work for fSSPUDE.

We note that, although this theoretical result is new, some existing MDP solvers use Equation 10 implicitly to cope with goal-oriented MDPs that have unavoidable dead ends. One example is the miniGPT package (Bonet and Geffner 2005); it allows the user to specify a value D and then uses it to implement Equation 10 in several algorithms including VI_{SSP} and LRTDP.

The Case of an Infinite Penalty

In contrast to fSSPUDE MDPs, no existing algorithm can solve iSSPUDE problems either implicitly or explicitly, so all algorithms for tackling these MDPs that we present in this section are completely novel.

Value Iteration for iSSPUDE MDPs

As for the finite-penalty case, we begin by deriving a Value Iteration-like algorithm for solving iSSPUDE. Finding a policy satisfying Equation 9 may seem hard, since we are effectively dealing with a multicriterion optimization problem. Note, however, the optimization criteria are, to a certain degree, independent — we can *first* find the set of policies whose probability of reaching the goal from s_0 is optimal, and *then* select from them the policy minimizing the expected cost of goal trajectories. This amounts to finding the optimal goal-probability function P^* first, then computing the optimal cost function $[J^*|P^*]$ conditional on P^*, and finally deriving an optimal policy from $[J^*|P^*]$. We consider these subproblems in order.

Finding P^*. The task of finding, for every state, the highest probability with which the goal can be reached by any policy in a given goal-oriented MDP has been studied before — it is a MAXPROB problem mentioned in the Back-

ground section. Solving a goal-oriented MDP according to the MAXPROB criterion means finding P^* that satisfies

$$P^*(s) = 1 \; \forall s \in \mathcal{G} \tag{11}$$
$$P^*(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} T(s,a,s') P^*(s') \; \forall s \notin \mathcal{G}$$

As already discussed, this P^* can be found by the VI_{MP} algorithm with an arbitrary initializing heuristic.

Finding $[J^*|P^*]$. We could derive optimality equations for calculating $[J^*|P^*]$ from first principles and then develop an algorithm for solving them. However, instead we present a more intuitive approach. Essentially, given P^*, we will build a modification M^{P^*} of the original MDP whose solution is exactly the cost function $[J^*|P^*]$. M^{P^*} will have no dead ends, have only actions greedy w.r.t. P^*, and have a transition function favoring transitions to states with higher probability of successfully reaching the goal. Crucially, M^{P^*} will turn out to be an SSP MDP, so we will be able to find $[J^*|P^*]$ with familiar machinery.

To construct M^{P^*}, observe that an optimal policy π^* for an iSSPUDE MDP, one whose cost function is $[J^*|P^*]$, must necessarily use only actions greedy w.r.t. P^*, i.e. those maximizing the right-hand side of Equation 11. For each state s, denote the set of such actions as $\mathcal{A}_s^{P^*}$. We focus on non-dead-end states, because for dead ends $[J^*|P^*](s) = 0$, and they will not be part of M^{P^*}. By Equations 11, for each such s, each $a^* \in \mathcal{A}_s^{P^*}$ satisfies the equality $P^*(s) = \sum_{s' \in \mathcal{S}} T(s,a^*,s') P^*(s')$. Note that this equation expresses the following relationship between event probabilities:

$$P\left(\begin{array}{c}\text{Goal was reached}\\ \text{from } s \text{ via optimal policy}\end{array}\right)$$
$$= \sum_{s' \in \mathcal{S}} P\left(\begin{array}{c}a^* \text{ caused}\\ s \to s' \text{ transition}\end{array} \bigwedge \begin{array}{c}\text{Goal was reached}\\ \text{from } s \text{ via optimal policy}\end{array}\right),$$

or, in a slightly rewritten form,

$$\sum_{s' \in \mathcal{S}} P\left(\begin{array}{c}a^* \text{ caused}\\ s \to s' \text{ transition}\end{array} \;\middle|\; \begin{array}{c}\text{Goal was reached}\\ \text{from } s \text{ via optimal policy}\end{array}\right) = 1,$$

where $P\left(\begin{array}{c}a^* \text{ caused}\\ s \to s' \text{ transition}\end{array} \;\middle|\; \begin{array}{c}\text{Goal was reached}\\ \text{from } s \text{ via optimal policy}\end{array}\right) = \frac{T(s,a^*,s')P^*(s')}{P^*(s)}$.

These equations essentially say that if a^* was executed in s and, as a result of following an optimal policy π^* the goal was reached, then with probability $\frac{T(s,a^*,s_1)P^*(s_1)}{P^*(s)}$ action a^* must have caused a transition from s to s_1, with probability $\frac{T(s,a^*,s_2)P^*(s_2)}{P^*(s)}$ it must have caused a transition to s_2, and so on. This means that if we want to find the vector $[J^*|P^*]$ of expected costs of goal-reaching trajectories under π^*, then it is enough to find the optimal cost function of MDP $M^{P^*} = \langle \mathcal{S}^{P^*}, \mathcal{A}^{P^*}, \mathcal{T}^{P^*}, \mathcal{C}^{P^*}, \mathcal{G}^{P^*}, s_0^{P^*} \rangle$, where $\mathcal{G}^{P^*}$ and $s_0^{P^*}$ (if known) are the same as $\mathcal{G}$ and s_0 for the iSSPUDE M that we are trying to solve; $\mathcal{S}^{P^*}$ is the same as $\mathcal{S}$ for M but does not include dead ends, i.e. states s for which $P^*(s) = 0$; $\mathcal{A}^{P^*} = \cup_{s \in \mathcal{S}} \mathcal{A}_s^{P^*}$, i.e. the set of actions consists of all P^*-greedy actions in each state; for each $a^* \in \mathcal{A}_s^{P^*}$, $\mathcal{T}^{P^*}(s,a^*,s') = \frac{\mathcal{T}(s,a^*,s')P^*(s')}{P^*(s)}$, as above, and a^* is "applicable" only in s; and $\mathcal{C}^{P^*}(s,a)$ is the same as $\mathcal{C}$ for M, except it is defined only for $a \in \mathcal{A}^{P^*}$.

As it turns out, we already know how to solve MDPs such as M^{P^*}:

Theorem 4. *For an iSSPUDE MDP M with $P^*(s_0) > 0$, MDP M^{P^*} constructed from M as above is an SSP MDP.*

Proof sketch. Indeed, M^{P^*} is "almost" like the original iSSPUDE MDP but has at least one proper policy because, by construction, it has no dead ends. □

Now, as we know (Bertsekas 1995), J^* for the SSP M^{P^*} satisfies $J^*(s) = \min_{a \in \mathcal{A}^{P^*}} \mathcal{C}^{P^*}(s,a) + \sum_{s' \in \mathcal{S}} \mathcal{T}^{P^*}(s,a,s') J^*(s')$. Therefore, by plugging in $\frac{\mathcal{T}(s,a,s')P^*(s')}{P^*(s)}$ in place of $\mathcal{T}^{P^*}(s,a,s')$ and $[J^*|P^*]$ in place of J^*, we can state the following theorem for the original iSSPUDE MDP M:

Theorem 5. *For an iSSPUDE MDP with the optimal goal-probability function P^*, the optimal cost function $[J^*|P^*]$ characterizing the minimum expected cost of trajectories that reach the goal satisfies*

$$[J^*|P^*](s) = 0 \; \forall s \text{ s.t. } P^*(s) = 0 \tag{12}$$
$$[J^*|P^*](s) = \min_{a \in \mathcal{A}^{P^*}} \left\{ \mathcal{C}(s,a) + \sum_{s' \in \mathcal{S}} \frac{\mathcal{T}(s,a,s')P^*(s')}{P^*(s)} [J^*|P^*](s') \right\}$$

Putting It All Together. Our construction not only let us derive the optimality equation for $[J^*|P^*]$, but also implies that $[J^*|P^*]$ can be found via VI, as in the case of SSP MDPs (Bertsekas 1995), over P^*-optimal actions and non-dead-end states. Moreover, since the optimal policy for an SSP MDP is greedy w.r.t. the optimal cost function and solving an iSSPUDE MDP ultimately reduces to solving an SSP, the following important result holds:

Theorem 6. *For every iSSPUDE MDP, there exists a Markovian deterministic policy π^* that can be derived from P^* and $[J^*|P^*]$ for non-dead-end states using*

$$\pi^*(s) = arg \min_{a \in \mathcal{A}^{P^*}} \left\{ \mathcal{C}(s,a) + \sum_{s' \in \mathcal{S}} \frac{\mathcal{T}(s,a,s')P^*(s')}{P^*(s)} [J^*|P^*](s') \right\} \tag{13}$$

Combining optimality equations 11 and 12 for P^* and $[J^*|P^*]$ respectively with Equation 13, we present a VI-based algorithm for solving iSSPUDE MDPs, called IVI (Infinite-penalty Value Iteration) in Algorithm 1.

Heuristic Search for iSSPUDE MDPs

As we established, solving an iSSPUDE MDP with VI is a two-stage process, whose first stage solves a MAXPROB MDP and whose second stage solves an SSP MDP. In the Background section we mentioned that both of these kinds of MDPs can be solved with heuristic search; MAXPROB — with the FRET framework, and SSP — with the FIND-AND-REVISE framework. This allows us to construct a heuristic search schema called SHS (Staged Heuristic Search) for iSSPUDE MDPs, presented in Algorithm 2.

Input: iSSPUDE MDP M
Output: Optimal policy π^* for non-dead-end states of M

1. Find P^* using arbitrarily initialized VI_{MP}.
2. Find $[J^*|P^*]$ using arbitrarily initialized VI_{SSP} over M^{P^*} with update equations 12

Return π^* derived from P^* and $[J^*|P^*]$ via Equation 13

Algorithm 1: IVI

Input: iSSPUDE MDP M
Output: Optimal policy $\pi^*_{s_0}$ for non-dead-end states of M rooted at s_0

1. Find $P^*_{s_0}$ using FRET initialized with an admissible heuristic $\hat{P} \geq P^*$
2. Find $[J^*|P^*]_{s_0}$ using FIND-AND-REVISE over M^{P^*} with optimality equations 12, initialized with an admissible heuristic $\hat{J} \leq [J^*|P^*]$.

Return $\pi^*_{s_0}$ derived from $P^*_{s_0}$ and $[J^*|P^*]_{s_0}$ via Equation 13

Algorithm 2: SHS

There are two major differences between Algorithms 1 and 2. The first one is that SHS produces functions $P^*_{s_0}$ and $[J^*|P^*]_{s_0}$ that are guaranteed to be optimal only over the states visited by some optimal policy $\pi^*_{s_0}$ starting from the initial state s_0. Accordingly, the SHS-produced policy $\pi^*_{s_0}$ specifies actions only for these states and does not prescribe any for other states. Second, SHS requires two admissible heuristics to find an optimal (partial) policy, one ($\hat{P}$) being an *upper* bound on P^* and the other ($\hat{J}$) being a *lower* bound on $[J^*|P^*]$.

Equivalences of Opimization Criteria

The presented algorithms for MDPs with unavoidable dead ends are significantly more complicated than those for MDPs with unavoidable ones. Nonetheless, intuition tell us that for a given tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C}, \mathcal{G}, D, s_0 \rangle$, solving it under the infinite-penalty criterion (i.e., as an iSSPUDE) should yield the same policy as solving it under the finite-penalty criterion (i.e., as an fSSPUDE) if in the latter case the penalty D is very large. Indeed, this can be stated as a theorem:

Theorem 7. *For iSSPUDE and fSSPUDE MDPs over the same domain, there exists the smallest finite penalty D_{thres} s.t. for all $D > D_{thres}$ the set of optimal policies of fSSPUDE (with penalty D) is identical to the set of optimal policies of iSSPUDE.*

Proof sketch. Although the full proof is technical, its main observation is simple — as D increases, it becomes such a large deterrent against hitting a dead end that any policy with a probability of reaching the goal lower than the optimal P^* starts having a higher expected cost of reaching the goal than policies optimal according to iSSPUDE's criterion. □

As a corollary, if we choose $D > D_{thres}$, we can be sure that at any given state s, all optimal ($\text{J}_F{}^*$-greedy) policies of the resulting fSSPUDE will have the same probability of reaching the goal, and this probability is $P^*(s)$ according to the infinite-penalty optimization criterion (and therefore will also have the same conditional expected cost $[J^*|P^*]$)

This prompts a question: what can we say about the probability of reaching the goal of $\text{J}_F{}^*$-greedy policies if we pick $D \leq D_{thres}$? Unfortunately, in this case different greedy policies may not only be suboptimal in terms of this probability, but even for fixed D each may have a different, arbitrarily low chance of reaching the goal. For example, consider an MDP with three states, s_0 (the initial state), d (a dead end), and g (a goal). Action a_d leads from s_0 to d with probability 0.5 and to g with probability 0.5 and costs 1 unit. Action a_g leads from s_0 to g also with probability 1, and costs 3 units. Finally, suppose we solve this MDP as an fSSPUDE with $D = 4$. It is easy to see that both policies, $\pi(s_0) = a_d$ and $\pi(s_0) = a_g$, have the same expected cost, 3. However, the former never reaches the goal with probability 0.5, while the latter always reaches it. The ultimate reason for this discrepancy is that the policy evaluation criterion of fSSPUDE is oblivious to policies's probability of reaching the goal, and optimizes for this parameter only indirectly, via policies' expected cost.

To summarize, we have two ways of finding an optimal policy in the infinite-penalty case, either by directly solving the corresponding iSSPUDE instance, or by choosing a sufficiently large D and solving the finite-penalty fSSPUDE MDP. We do not know of a principled way to choose it, but it istypically easy to guess by inspecting the MDP. Thus, although the latter method gives no a-priori guarantees, it often yields a correct answer in practice.

Experimental Results

The objective of our experiments was to find out the most practically efficient way of finding the optimal policy in the presence of unavoidable dead ends and infinite penalty for visiting them, by solving an iSSPUDE MDP or an fSSPUDE MDP with a large D. To make a fair comparison between these methods, we employ very similar algorithms to handle them. For both classes, the most efficient optimal solution methods are heuristic search techniques, so in our experiments we assume knowledge of the initial state and use only algorithms of this type.

To solve an fSSPUDE we use the implementation of the LRTDP algorithm, an instance of the FIND-AND-REVISE heuristic search framework for SSPs, available in the miniGPT package (Bonet and Geffner 2005). As a source of admissible heuristic state costs/goal-probability values, we choose the maximum of atom-min-forward heuristic (Haslum and Geffner 2000) and SixthSense (Kolobov et al. 2011). The sole purpose of the latter is to soundly identify many of the dead-end states and assign the value of D to them. (Identifying a state as a dead end may be nontrivial if the state has actions leading to other states.)

Since solving iSSPUDE involves tackling two MDPs, a MAXPROB and an SSP, to instantiate the SHS schema (Algorithm 2) we use two heuristic search algorithms. For the MAXPROB component, we use a specially adapted version of the same LRTDP implementation (Kolobov et al. 2011) as an example of the FRET framework, equipped with SixthSense (note that the atom-min-forward heuristic is cost-based and does not apply to MAXPROB MDPs). For the SSP component, we use LRTDP from miniGPT, as for fSSPUDE, with atom-min-forward; SixthSense is unnecessary because SSP has no dead ends.

Our benchmarks were problems 1 through 6 of the Exploding Blocks World domain from IPPC-2008 (Bryce and Buffet 2008) and problems 1 through 15 of the Drive domain from IPPC-06 (Buffet and Aberdeen 2006). Most problems in both domains have unavoidable dead ends. To set the D penalty for the fSSPUDE model, we examined each problem and tried to come up with an intuitive, easily justifiable value for it. For all problems, solving the fSSPUDE with $D = 500$ yielded a policy that was optimal under both the finite-penalty and infinite-penalty criterion.

Solving the fSSPUDE with $D = 500$ and iSSPUDE versions of each problem with the above implementations yielded the same qualitative outcome on all benchmarks. In terms of speed, solving fSSPUDE was at least an order of magnitude faster than solving iSSPUDE. The difference in used memory was occasionally smaller, but only because both algorithms visited nearly the entire state space reachable from s_0 on some problems. Moreover, in terms of memory as well as speed the difference between solving fSSPUDE and iSSPUDE was the largest (that is, solving iSSPUDE was comparatively the *least* efficient) when the given MDP had $P^*_{s_0}(s) = 1$, i.e. the MDP had no dead ends at all or had only avoidable ones.

Although seemingly surprising, these performance patterns have a fundamental reason. Recall that FRET algorithms, used for solving the MAXPROB part of an iSSPUDE, use the BET operator. BET, for every encountered fixed point $P^\times$ of the Bellman backup operator needs to traverse the transition graph involving all actions greedy w.r.t. $P^\times$, starting from s_0. Also, FRET needs to be initialized with an admissible heuristic, in our experiments – SixthSense, which assigns the value of 0 to states it believes to be dead ends and 1 to the rest.

Now, consider how FRET operates on a MAXPROB corresponding to an iSSPUDE instance that does not have any dead ends, i.e. on the kind of iSSPUDE MDPs that, as our experiments show, is most problematic. For such a MAXPROB, there exists only one admissible heuristic function, $\hat{P}(s) = 1$ for all s, because $P^*(s) = 1$ for all s and an admissible $\hat{P}$ needs to satisfy $\hat{P}(s) \geq P^*(s)$ everywhere. Thus, the heuristic FRET starts with is actually the optimal goal-probability function, and as a consequence, is a fixed point of the Bellman backup operator. Therefore, to conclude that $\hat{P}$ is optimal, FRET needs build its greedy transition graph. Observe, however, that since $\hat{P}$ is 1 everywhere, this transition graph includes every state reachable from s_0, and uses every action in the MDP! Traversing it is very expensive, and forces FRET to enumerate the entire reachable state space of the problem.

The same performance bottleneck, although to a lesser extent, can also be observed on iSSPUDE instances that do have unavoidable dead ends. Building large transition graphs significantly slows down FRET (and hence, SHS) even when P^* is far from being 1 everywhere.

The above reasoning may explain why solving iSSPUDE is slow, but by itself does not explain why solving fSSPUDE is fast in comparison. For instance, we might expect the performance of FIND-AND-REVISE algorithms on fSSPUDE to suffer in the following situations. Suppose state s is a dead end not avoidable from s_0 by any policy. This means that $J^*(s) = D$ under the finite-penalty optimization criterion, and that s is reachable from s_0 by any optimal policy. Thus, FIND-AND-REVISE will halt no earlier than the cost of s under the current cost function reaches D. Moreover, suppose that the heuristic $\hat{J}$ initializes the cost of s to 0 — this is one of the possible admissible costs for s. Finally, assume that all actions in s lead back to s with probability 1 and cost 1 unit. In such a situation, an FIND-AND-REVISE algorithm will need to update the cost of s D times before convergence. Clearly, this will make the performance of FIND-AND-REVISE very bad is the chosen value of D is very large. This raises the question: was solving fSSPUDE in the above experiments so much more efficient than solving iSSPUDE due to our choice of (a rather small) value for D?

To dispel these concerns, we solved fSSPUDE instances of the aforementioned benchmarks with $D = 5 \cdot 10^8$ instead of 500. On all of the 21 problems, the increase in speed compared to the case of fSSPUDE with $D = 500$ was no more than a factor of 1.5. The reason for such a small discrepancy is the fact that, at least on our benchmarks, FIND-AND-REVISE almost never runs into the pathological case described above thanks to the atom-min-forward and SixthSense heuristics. They identify majority of dead ends encountered by LRTDP and immediately set their costs to D. Thus, instead of spending many updates on such states, LRTDP gets their optimal costs in just one step. To test this explanation, we disabled these heuristics and assigned the cost of 0 to all states at initialization. As predicted, the solution time of the fSSPUDE instances skyrocketed by orders of magnitude.

The presented results appear to imply an unsatisfying fact — on iSSPUDE MDPs that are SSPs, the presented algorithms for solving iSSPUDE are not nearly as efficient as algorithmic schema for SSPs, such as FIND-AND-REVISE. The caveat, however, is that the price SSP algorithms pay for efficiency is *assuming* the existence of proper solutions. iSSPUDE algorithms, on the other hand, implicitly *prove* the existence of such a solution, and are therefore theoretically more robust.

Conclusion

A significant limitation of Stochastic Shortest Path MDPs is their inability to model dead-end states, consequences of catastrophic action outcomes that make reaching the goal

impossible. While attempts to incorporate dead ends into SSP have been made before, a principled theory of goal-oriented MDPs with dead-end states has been lacking.

In this paper, we present new general MDP classes that subsume MDPs and make increasingly weaker assumptions about the presence of dead ends. SSPADE assumes that dead ends are present but an agent can avoid them if it acts optimally from the initial state. fSSPUDE admits unavoidable dead ends but expects that an agent can put a finite price on running into a dead end. iSSPUDE MDPs model scenarios in which entering a dead end carries an infinite penalty and is to be avoided at all costs.

For these MDP classes we present VI-based and heuristic search algorithms. We also study the conditions under which they have equivalent solutions. Our empirical results show that, in practice, solving fSSPUDE is much more efficient and yields the same optimal policies as iSSPUDE.

In the future we hope to answer the question: are iSSPUDE MDPs fundamentally harder than fSSPUDE, or can we invent more efficient heuristic search algorithms for them?

Acknowledgments. This work has been supported by NSF grant IIS-1016465, ONR grant N00014-12-1-0211, and the UW WRF/TJ Cable Professorship.

References

Bellman, R. 1957. *Dynamic Programming*. Princeton University Press.

Bertsekas, D. 1995. *Dynamic Programming and Optimal Control*. Athena Scientific.

Bonet, B., and Geffner, H. 2003a. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *IJCAI*, 1233–1238.

Bonet, B., and Geffner, H. 2003b. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *ICAPS'03*, 12–21.

Bonet, B., and Geffner, H. 2005. mGPT: A probabilistic planner based on heuristic search. *Journal of Artificial Intelligence Research* 24:933–944.

Bryce, D., and Buffet, O. 2008. International planning competition, uncertainty part: Benchmarks and results. In *http://ippc-2008.loria.fr/wiki/images/0/03/Results.pdf*.

Buffet, O., and Aberdeen, D. 2006. The factored policy gradient planner (ipc-06 version). In *Fifth International Planning Competition at ICAPS'06*.

Chatterjee, K.; Majumdar, R.; and Henzinger, T. A. 2006. Markov decision processes with multiple objectives. In *Proceedings of Twenty-third Annual Symposium on Theoretical Aspects of Computer Science*, 325–336.

Haslum, P., and Geffner, H. 2000. Admissible heuristic for optimal planning. In *AIPS*, 140149.

Keyder, E., and Geffner, H. 2008. The HMDPP planner for planning with probabilities. In *Sixth International Planning Competition at ICAPS'08*.

Kolobov, A.; Mausam; Weld, D.; and Geffner, H. 2011. Heuristic search for Generalized Stochastic Shortest Path MDPs. In *ICAPS'11*.

Kolobov, A.; Mausam; and Weld, D. 2010. SixthSense: Fast and reliable recognition of dead ends in MDPs. In *AAAI'10*.

Little, I., and Thiebaux, S. 2007. Probabilistic planning vs. replanning. In *ICAPS Workshop on IPC: Past, Present and Future*.

Wakuta, K. 1995. Vector-valued Markov decisionprocesses and the systems of linear inequalities. *Stochastic Processes and their Applications* 56:159–169.

Structural Patterns Beyond Forks: Extending the Complexity Boundaries of Classical Planning

Michael Katz
Saarland University
Saarbrücken, Germany
katz@cs.uni-saarland.de

Emil Keyder
INRIA
Nancy, France
emilkeyder@gmail.com

Abstract

Tractability analysis in terms of the causal graphs of planning problems has emerged as an important area of research in recent years, leading to new methods for the derivation of domain-independent heuristics (Katz and Domshlak 2010). Here we continue this work, extending our knowledge of the frontier between tractable and NP-complete fragments. We close some gaps left in previous work, and introduce novel causal graph fragments that we call the *hourglass* and *semi-fork*, for which under certain additional assumptions optimal planning is in P. We show that relaxing any one of the restrictions required for this tractability leads to NP-complete problems. Our results are of both theoretical and practical interest, as these fragments can be used in existing frameworks to derive new abstraction heuristics. Before they can be used, however, a number of practical issues must be addressed. We discuss these issues and propose some solutions.

Introduction

Quantifying the complexity of classical planning problems in terms of their structure has long been an important research problem. Recent work in this area has focused on *causal graphs* (Domshlak and Dinitz 2001; Brafman and Domshlak 2003; Chen and Giménez 2008; Katz and Domshlak 2008; 2010; Giménez and Jonsson 2008), directed graphs whose nodes represent the variables of the problem and whose edges give information about dependencies between variables (Knoblock 1994). Combining limitations on causal graph structure with further restrictions on the sizes of variable domains and k-dependence, defined as the maximum number of variables on which an action has preconditions while not changing their values, has led to complexity results that apply to a wide range of problems (Katz and Domshlak 2008; Giménez and Jonsson 2009). Such results are not of purely theoretical interest, as the causal graph is used in a variety of practical applications from problem decomposition (Brafman and Domshlak 2006) to the derivation of non-admissible domain-independent heuristics for satisficing planning (Helmert 2004).

The work we present here is motivated by a different use of tractable fragments of the causal graph: the derivation of *admissible* domain-independent heuristics. Search with such heuristics is one of the most successful approaches to optimal planning, and an important advance in this field over the last few years has been the introduction of *structural pattern heuristics* (Katz and Domshlak 2010). The idea behind these heuristics is to project planning problems onto fragments of causal graphs known to be tractable for optimal planning, and to use the costs of solutions to these as guidance for the original problem. Structural pattern heuristics play an important theoretical role in optimal planning, as they represent one of the handful of existing ideas for deriving admissible heuristics (Helmert and Domshlak 2009).

The usefulness of structural pattern heuristics increases directly with the availability of causal graph fragments that are known to be solvable optimally in polynomial time. Until now, they have made use of two non-trivial structures known as the *fork* and the *inverted fork*. Our principal aim here is to discover the limits of tractability for these two structures, removing restrictions and considering wider classes of causal graphs until the point at which optimal planning becomes NP-complete is found. This approach allows us to close several gaps in previous work, and results in the introduction of two new classes that under certain limitations are tractable for optimal planning and can be used in such heuristics, *hourglasses* and *semiforks*. We also show that the relaxation of any one of the assumptions required for this tractability leads to an NP-complete problem. While the use of these classes in structural pattern heuristics could improve their estimates, a number of practical issues remain to be solved before they can be adapted to that context. We briefly discuss these issues, and propose some solutions.

Preliminaries

We consider planning problems in the SAS$^+$ formalism (Bäckström and Nebel 1995), given by a quintuple $\Pi = \langle V, A, I, G, \mathit{cost} \rangle$ where:

- V is a set of *state variables*, each $v \in V$ associated with a finite domain $\mathcal{D}(v)$. The value assigned to a variable v by a (possibly partial) assignment p to V is denoted by $p[v]$. A complete assignment s to V is called a *state*, and the set of all possible complete assignments S is the *state space* of Π. I is the *initial state*. The *goal* G is a partial assignment to V; a state s is a *goal state* iff $G \subseteq s$.
- A is a finite set of *actions*, each action $a \in A$ given by a pair $\langle \mathsf{pre}(a), \mathsf{eff}(a) \rangle$ of partial assignments to V called *preconditions* and *effects*, respectively. By $A_v \subseteq A$, we

denote the actions changing the value of v. $cost : A \to \mathbb{R}^{0+}$ is a real-valued, non-negative *cost* function.

An action a is applicable in a state s iff $\mathsf{pre}(a) \subseteq s$. The state s' resulting from applying a in s is denoted by $s[\![a]\!]$ and differs from s in that $s[v] = \mathsf{eff}(a)[v]$ whenever this is defined. $s[\![\langle a_1, \ldots, a_k \rangle]\!]$ denotes the state resulting from sequential application of the actions $a_1, \ldots, a_k$ in s. Such an action sequence is an *s-plan* if $G \subseteq s[\![\langle a_1, \ldots, a_k \rangle]\!]$, and it is an *optimal* s-plan if the summed cost $\sum_{i=1}^{k} cost(a_i)$ is minimal among all s-plans. The aim of (optimal) planning is to find an (optimal) I-plan. In what follows, we denote a plan for state s with $\pi(s)$ or just π when s is clear from the context, and use the notation π^* to specify that a plan is optimal. h^* denotes the cost of such an optimal plan.

The *causal graph* of Π is a digraph $CG(\Pi) = \langle V, E \rangle$ over the set of nodes V that contains an arc (v, v') iff $v \neq v'$ and there exists $a \in A$ such that $\mathsf{eff}(a)[v']$ and either $\mathsf{pre}(a)[v]$ or $\mathsf{eff}(a)[v]$ is specified. Given a variable v, we use the shorthands $\mathsf{pred}(v) = \{v' \mid (v', v) \in E\}$ and $\mathsf{succ}(v) = \{v' \mid (v, v') \in E\}$. The *domain transition graph* $DTG(\Pi, v)$ of $v \in V$ is an arc-labeled digraph with nodes $\mathcal{D}(v)$ that contains an arc (ϑ, ϑ') labeled with $\mathsf{pre}(a) \setminus \mathsf{pre}(a)[v]$ iff $\mathsf{eff}(a)[v] = \vartheta'$ and either $\mathsf{pre}(a)[v] = \vartheta$ or $\mathsf{pre}(a)[v]$ is unspecified.

In this paper we extend two previously studied causal graph structures known as the *fork* and *inverted fork*. These structures are digraphs $G = (N, E)$ such that there exists a node $r \in N$ for which $(u, v) \in E \iff u = r$, if the structure is a fork, and $(u, v) \in E \iff v = r$, if the structure is an inverted fork. We refer to planning problems whose causal graphs are (inverted) forks as (inverted) fork structured planning problems. Optimal planning has been shown to be in P for fork structured planning problems if $|\mathcal{D}(r)| = 2$, and for inverted fork structured planning problems for any $|\mathcal{D}(r)| \in O(1)$ (Katz and Domshlak 2010).

Forks

We start by closing the gap left by Katz and Domshlak (2010) in the complexity of cost-optimal planning for fork-structured tasks:

Theorem 1 *Cost-optimal planning for fork structured problems with causal graph rooted in a ternary-valued variable is NP-complete.*

Proof: Membership in NP is obvious. The proof of hardness is by reduction from the shortest common superstring problem (SCS). Let $x_1, \ldots, x_n$ be a set of strings over a binary alphabet. Given x_i, let x'_i denote the string over the alphabet $\{0, 1, 2\}$ that results from inserting the symbol 2 at the beginning, end, and between each pair of symbols in x_i. There then exists an SCS of length k for $x_1, \ldots, x_n$ iff there exists an SCS of length $2k + 1$ for $x'_1, \ldots, x'_n$.

Given a planning problem $\Pi = \langle V, A, I, G, cost \rangle$, where:

- $V = \{r, y_1, \ldots, y_n\}$, with $\mathcal{D}(r) = \{0, 1, 2\}$ and $\mathcal{D}(y_i) = \{0, \ldots, |x'_i|\}$ for $i = 1, \ldots, n$,
- $A = \{a_{ij} \mid i = 1, \ldots, n, j = 0, \ldots, |x'_i| - 1\} \cup \{r_{0\to2}, r_{2\to0}, r_{1\to2}, r_{2\to1}\}$, where $a_{ij} = \langle \{y_i{=}j, r{=}x'_i[j]\}, \{y_i{=}j+1\} \rangle$, in which $x'_i[j]$ denotes the jth symbol of x'_i, $r_{\alpha\to\beta} = \langle \{r{=}\alpha\}, \{r{=}\beta\} \rangle$, $cost(a_{ij}) = 0$ for all a_{ij} and $cost(r_{\alpha\to\beta}) = 1$,
- $I = \{r{=}2\} \cup \{y_i{=}0 \mid i = 1, \ldots, n\}$, and
- $G = \{r{=}2\} \cup \{y_i{=}|x'_i| \mid i = 1, \ldots n\}$,

finding an optimal plan for Π is equivalent to finding an SCS for $x'_1, \ldots, x'_n$. The causal graph of Π is a fork with root r and leaves $y_1, \ldots y_n$. The DTG for the variable r is a chain with 3 nodes, with the value 2 at the center doubly connected to each of the values 0, 1, at the two sides (Figure 1). The DTG for each of the variables $y_1, \ldots, y_n$ is a chain in which there is a single path that traverses the values of y_i in ascending order, and that requires for each transition that the variable r have the value corresponding to that position in the string x'_i (Figure 2).

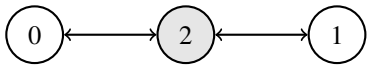


Figure 1: DTG for variable r.

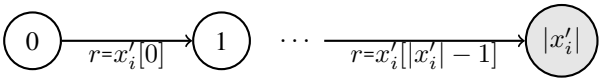


Figure 2: DTG for variable y_i.

Since the variables y_i can transition to their next values only when r has the value of the corresponding position in the string x'_i, the sequence of values taken on by the variable r must correspond to a superstring of the set of strings $\{x'_0, \ldots, x'_n\}$. The only actions with non-zero cost are those that change the value of r, and there therefore exists a plan for Π with cost $2k$ iff there exists a superstring of $\{x'_0, \ldots, x'_n\}$ with length $2k + 1$, and a superstring of $\{x_0, \ldots, x_n\}$ with length k. As this transformation can be performed in polynomial time, this shows the desired result. ∎

Unfortunately, this does not shed light on the complexity of deciding plan existence. Our next result concerns this problem for fork-structured planning problems where a more general property holds for the DTG of the root variable:

Theorem 2 *Let Π be a planning task with a fork-structured causal graph rooted at variable r, and let $\mathcal{G}$ be the condensed graph of $DTG(\Pi, r)$, with one node for each strongly connected component (SCC) of $DTG(\Pi, r)$. Plan existence for Π can be decided in polynomial time if $\mathcal{G}$ has only a polynomial number of paths.*

Proof: Consider a (necessarily cycle-free, as the condensed graph is directed acyclic) path $P_1, \ldots, P_m$ in $\mathcal{G}$, where each node P_i corresponds to a *set* of values of r that make up an SCC in $DTG(\Pi, r)$. For $0 \leq i \leq m$ and for $v \in \mathsf{succ}(r)$, we define the sets C_v^i inductively as follows:

- $C_v^0 = \{I[v]\}$, and

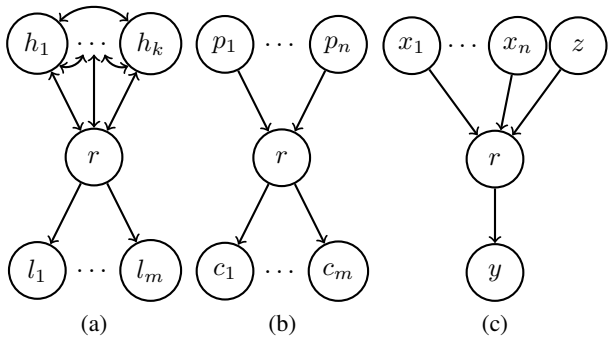


Figure 3: (a) Semifork and (b) hourglass causal graphs. (c) Causal graph structure for reduction of Theorem 7.

- for $i > 0$, C_v^i is the set of all values in $\mathcal{D}(v)$ achievable from any value in C_v^{i-1} using actions in A_v that have preconditions only on values of r that make up the SCC corresponding to P_i.

Note that it follows from this definition that C_v^i grows monotonically in i, i.e. $C_v^{i-1} \subseteq C_v^i$ for all i. Given a path $P_1, \ldots, P_m$ in $\mathcal{G}$, if for all $v \in \mathsf{succ}(r)$ we have $G[v] \in C_v^m$, and $G[r] \in P_m$, then a plan for Π can be constructed from the above in polynomial time. Π is solvable iff there exists a (cycle-free) path $P_1, \ldots, P_m$ in the condensed graph $\mathcal{G}$ such that $G[r] \in P_m$ and $G[v] \in C_v^m$ for all $v \in \mathsf{succ}(r)$. Since there are a polynomial number of paths to check, this proves the result. ∎

We note that when $|\mathcal{D}(r)| = O(1)$, the condensed graph of $DTG(\Pi, r)$ has only $O(1)$ paths, and Theorem 2 is applicable. This result therefore implies that plan existence for fork-structured tasks with constant bounded root domains is in P and closes the gap left by Domshlak and Dinitz (2001).

Semifork Causal Graphs

We now explore a graph structure that we call a *semifork*:

Definition 1 (Semifork) *A digraph $G = (N, E)$ is a* semifork *if there exists a set of nodes $L \subset N$, $L \neq \emptyset$ such that (i) $\forall v \in L$ outdegree$(v) = 0$, and (ii) there exists a node $r \in N \setminus L$ such that $(u, v) \in E$ and $v \in L$ imply $u = r$.*

Informally, one part of a semifork causal graph has fork structure, and the remaining nodes have edges only among themselves or to the root of the fork (Figure 3a). We refer to the node r as the *center* of the causal graph, the nodes L as the semifork's *leaves*, and the rest of the nodes $N \setminus (L \cup \{r\})$ as the semifork's *hat*. Note that given a graph G, there may be multiple possibilities for choosing L that result in different interpretations of G as a semifork.[1] We now show a tractability result for semifork structured causal graphs, extending a previous result by Katz and Domshlak (2010):

[1]Each subset of the child nodes of a fork induces a different semifork when used as L, for example.

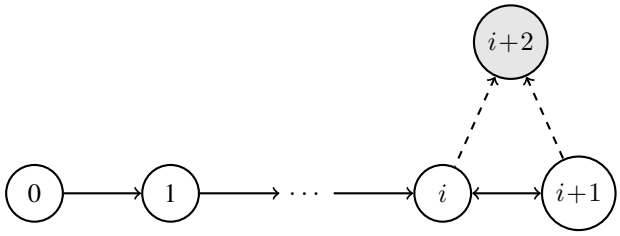


Figure 4: DTG for variable r_i in Π_{*i} (lower). Transitions represented with dashed edges may be present or not depending on the goal value defined for r or lack thereof.

Theorem 3 (Tractable Semiforks) *Given a constant k and a semifork-structured planning task $\Pi = \langle V, A, I, G, cost \rangle$ with center $r \in V$, $|\mathcal{D}(r)| = 2$, and $|\mathsf{hat}| < k$, cost-optimal planning for Π is polynomial in $\|\Pi\|^k$.*

Proof: We note that given a sequence of changes to r, the hat and fork portions of the planning problem can be decoupled and solved separately. Let Π^h denote the planning problem that results from removing all leaf variables from the problem, and $\pi(h)_i^*$ a cost minimal plan among the plans for Π^h in which the value of r is changed *at least* i times. In turn, let Π^f denote the problem in which all hat variables are removed and the value of r can be changed with no preconditions and cost 0, and $\pi(f)_i^*$ a cost minimal plan among the plans that set all the leaf variables to their goal values while changing the value of r *at most* i times. Any optimal plan π^* for Π can be partitioned into two such cost-minimal plans[2] by choosing i to be the number of changes to r in π^*. The optimal plan for Π can therefore be found by considering cost-minimal plans for Π^h and Π^f for each possible i:

$$cost(\pi^*(\Pi)) = \min_i [cost(\pi(h)_i^*) + cost(\pi(f)_i^*)]$$

and interleaving the actions of the two plans as required. Note that if (i) given a value of i, both $\pi(h)_i^*$ and $\pi(f)_i^*$ can be obtained in polynomial time, and (ii) there is an upper bound b on i that is polynomial in $\|\Pi\|$ such that both $cost(\pi(h)_i^*)$ and $cost(\pi(f)_i^*)$ are non-decreasing for $i > b$, the semifork problem can also be solved optimally in polynomial time. For $cost(\pi(h)_i^*)$, any bound will do, as increasing the value of i can only exclude plans making fewer changes to r. For $cost(\pi(f)_i^*)$, this bound is given by $b = \max_{v \in \mathsf{leaves}(r)} |\mathcal{D}(v)| + 1$ (Katz and Domshlak 2010). We now proceed to the formal description of how to obtain $\pi(h)_i^*$ and $\pi(f)_i^*$ in polynomial time.

We first describe the construction of a planning problem Π_i^h for $i \geq 1$, whose optimal plans correspond to optimal plans $\pi(h)_i^*$. Assuming wlog that $I[r] = 0$, we restrict Π to the variables $\mathsf{hat} \cup \{r\}$, while modifying the DTG of r to consist of $i + 3$ values (Figure 4):

- $V_i = \mathsf{hat} \cup \{r_i\}$, with $\mathcal{D}(r_i) = \{0, \ldots, i + 2\}$
- $$A_i = \bigcup_{v \in \mathsf{hat}} A_v \cup \bigcup_{j=0}^{i+1} A^j \cup A^g$$

[2]Otherwise, each could be independently replaced with any cost-minimal plan.

where $A^g = \{a_i^g, a_{i+1}^g\}$ if no goal value is defined for r, $A^g = \{a_i^g\}$ if $G[r] + i$ is even, and $A^g = \{a_{i+1}^g\}$ if $G[r] + i$ is odd, where $a_j^g = \langle\{r_i{=}j\}, \{r_i{=}i+2\}\rangle$. For $0 \leq j \leq i$,

$$A^j = \bigcup_{a \in A_r} \left\{ a_f \;\middle|\; \begin{array}{l} \mathsf{pre}(a_f)[r_i] = j, \mathsf{eff}(a_f)[r_i] = j+1, \\ \mathsf{pre}(a)[r] + j \text{ is even, and} \\ \mathsf{pre}(a_f)[v] = \mathsf{pre}(a)[v] \text{ and} \\ \mathsf{eff}(a_f)[v] = \mathsf{eff}(a)[v] \quad \forall v \in \mathsf{hat} \end{array} \right\},$$

and for $j = i + 1$,

$$A^{i+1} = \bigcup_{a \in A_r} \left\{ a_b \;\middle|\; \begin{array}{l} \mathsf{pre}(a_b)[r_i] = i+1, \mathsf{eff}(a_b)[r_i] = i, \\ \mathsf{pre}(a)[r] + i + 1 \text{ is even, and} \\ \mathsf{pre}(a_b)[v] = \mathsf{pre}(a)[v] \text{ and} \\ \mathsf{eff}(a_b)[v] = \mathsf{eff}(a)[v] \quad \forall v \in \mathsf{hat} \end{array} \right\},$$

and $cost_i(a_f) = cost(a)$, $cost_i(a_b) = cost(a)$, $cost_i(a_g^i) = cost_i(a_g^{i+1}) = 0$,

- $I_i[v] = I[v]$ for $v \in \mathsf{hat}(r)$ and $I_i[r_i] = 0$, and
- $G_i[v] = G[v]$ for $v \in \mathsf{hat}(r)$ and $G_i[r_i] = i + 2$.

Note that due to the requirement that $\mathsf{pre}(a)[r]{+}j$ be even, actions preconditioned by $r{=}0$ appear in A^j only for even j and those preconditioned by 1 for odd j. In order to reach the goal value of r_i, the plan must apply a sequence of actions that change r i times, and can then alternate between the values i and $i + 1$ before achieving the goal, preconditioned on the original goal value of r. Since the task Π_i^h has at most k variables, it is solvable optimally in polynomial time, and a cost-minimal plan $\pi(h)_i^*$ can be obtained by replacing the actions in an optimal plan for Π_i^h with the corresponding actions from A, that is, replacing r_i-changing actions with their r-changing originals.

We now consider how to obtain the plans $\pi(f)_i^*$. Given a sequence of value changes of the variable r, all children $c_j \in \mathsf{leaves}(r)$ are independent of each other and of the hat. Provided a number i of value changes for r, a cost-minimal plan for each child variable can therefore be obtained in polynomial time, and these plans can be interleaved to obtain a cost minimal plan.[3]

In order to obtain an optimal plan for Π, it is therefore sufficient to iterate over all values $0 \leq i \leq b$, where $b = \max_{v \in \mathsf{leaves}(r)} |\mathcal{D}(v)| + 1$, and store the plans that result in the cheapest summed cost $\pi(h)_i^* + \pi(f)_i^*$. These plans can then be interleaved by adding the actions in $\pi(f)_i^*$ at the earliest possible point during the execution of $\pi(h)_i^*$ to obtain an optimal plan. ∎

Relaxing the constant bound on the size of hat makes even the plan existence problem NP-complete, as arbitrary planning problems can then be encoded. The same is the case when the binary bound on the domain size of the center variable is relaxed:

[3]For further detail see the proof of Theorem 4 by Katz and Domshlak (2010).

Theorem 4 *Plan existence for semifork structured problems with* $|\mathsf{hat}| = 1$ *and center variable domain size* ≥ 3 *is NP-complete.*

Proof: The idea behind the proof is similar to that of Theorem 1. Given a set of strings over a binary alphabet and a parameter k, we construct a planning problem in the same way as we did there, except with an additional variable x on which all actions that change the value of r have a prevail condition. The causal graph of this problem is then a semifork with a single variable in the hat. The domain transition graph of x is a chain of length $2k$, alternating values of which allow transitions in r from 0 or 1 to 2 and from 2 to 0 or 1, respectively. This variable enforces that the value of r can be changed from 2 to either 0 or 1 and then back to 2 at most k times, and as before the problem is then solvable iff there exists a superstring of the set of strings of length k. ∎

Hourglass Causal Graphs

We now introduce a digraph structure that we call the *hourglass* (Figure 3b):

Definition 2 (Hourglass) *A digraph* $G = (N, E)$ *is an hourglass if (i)* $(u, v) \in E$ *implies* $(v, u) \notin E$*, and (ii) there exists a node* $r \in N$*, such that for each* $(u, v) \in E$*, either* $u = r$ *or* $v = r$*.*

We call the node r the *center* of the graph. We refer to its predecessor nodes $\mathsf{pred}(r) = \{u \in N \mid (u, r) \in E\}$ as *parents*, and its successor nodes $\mathsf{succ}(r) = \{v \in N \mid (r, v) \in E\}$ as *children*. Intuitively, the hourglass differs from the semifork in that edges between the parent nodes are not allowed, and the outgoing edges of the center node all lead to child variables. We begin with the positive result that imposing a constant domain bound on the center and the child variables makes optimal planning tractable:

Theorem 5 (Hourglass with bound on child domain size) *Given a constant* d *and an hourglass-structured planning task* Π *with center variable domain size* $|\mathcal{D}(r)| \leq d$*, and* $|\mathcal{D}(c_i)| \leq d$ *for all child variables* $c_i \in \mathsf{succ}(r)$*, optimal planning for* Π *is polynomial in* $\|\Pi\|^k$*, where* $k = d^{(d^2+2)}$*.*

Proof: First, we note that the bound d on the domain size of the child variables also constitutes a bound on the length of the sequence of prevail values required from r for any one child. Considering also that up to d intermediate values of r may be required in moving from one value to another, the total length of the sequence of r values for a single child is d^2. The number of all possible sequences of that length is d^{d^2}, and a (loose) upper bound on the length of a sequence that contains all such sequences as subsequences is given by $k = d^2 \cdot d^{d^2} = d^{(d^2+2)}$. The number of possible r-changing action sequences that can achieve these values is then a polynomial $|A|^k$. Given such an action sequence, an optimal sequence of actions for the parent variables that satisfies all the required preconditions can be found in linear time. It is

therefore sufficient to check each possible sequence of actions up to length k and choose the one that results in the globally optimal plan. ∎

However, when such a bound is not imposed, even satisficing planning quickly becomes NP-complete:

Theorem 6 *Satisficing planning for hourglasses with center variable domain size ≥ 3 is NP-complete.*

This follows trivially from the proof of Theorem 4, as the problem in the proof has hourglass structure. Bounding the domain sizes of the child variables without bounding that of the center variable does not help either, as it follows from results for inverted forks by Domshlak and Dinitz (2001) that satisficing planning in this case is NP-complete.

We now consider the complexity of planning for problems with hourglass causal graphs with the added parameter of k-*dependence* (Katz and Domshlak 2008):

Definition 3 (k-dependent) *An action a is k-dependent if the size of its prevail condition, that is the number of variables that it has preconditions on but whose values it does not change, is $\leq k$. A planning problem Π is k-dependent if all its actions are k-dependent.*

We first show that for 2-dependent hourglass-structured problems even satisficing planning is NP-complete:

Theorem 7 (2-dependent hourglass) *Plan existence for the 2-dependent hourglass problem with center variable domain size 2 is NP-complete.*

Proof: Membership in NP is obvious, we show hardness by a polynomial reduction from SAT. Let $P = (C, U)$ be a SAT problem with m clauses $C = \{C_0, \ldots, C_{m-1}\}$ and n variables $U = \{u_1, \ldots, u_n\}$. We construct an hourglass problem Π with a single child variable y and $n + 1$ parent variables $x_1, \ldots, x_n, z$ (Figure 3c). The goal of the problem is defined only for the child variable y, and its purpose is to force the value of the center variable to change exactly $2m-1$ times. Its DTG is therefore an ascending chain of length $2m$ with values $0, \ldots, 2m-1$, transitions $i \rightarrow i+1$ that require alternating values of r beginning with r=1, and goal $2m-1$. A solution to Π then exists iff the parent nodes of the problem permit the value of r to be changed $2m-1$ times. The parent variables $x_1, \ldots, x_n$ correspond to the variables of the SAT problem, and have DTGs that allow their values to be set once to either 0 or 1, from an initial "undefined" value. The variable z has a DTG which consists of a chain with $2m$ values, whose even values $2i$, in conjunction with a value for some variable appearing in C_i that satisfies it, allow r to be set to 1, and whose odd values allow r to be set to 0. To solve the problem, a plan must set the values of the x_i variables to appropriate values, and advance through the DTG of Z while setting alternating values for r.

Formally, we define $\Pi = \langle V, A, I, G, cost \rangle$ as follows:

- $V = \{x_1, \ldots, x_n, z, r, y\}$
- $I = \{r{=}0, x_1{=}\bot, \ldots, x_n{=}\bot, z{=}0, y{=}0\}$
- $G = \{y{=}2m-1\}$
- $$A = \bigcup_{i=1}^{n} A_{x_i} \cup \bigcup_{i=0}^{2m-1} \{a_y^i\} \cup \bigcup_{i=0}^{2m-3} \{a_z^i\} \cup A_{r\rightarrow 0} \cup \bigcup_{i=0}^{m-1} A_{r\rightarrow 1}^i$$

  where
  - $A_{x_i} = \{\langle \{x_i{=}\bot\}, \{x_i{=}0\} \rangle, \langle \{x_i{=}\bot\}, \{x_i{=}1\} \rangle\}$,
  - $a_y^i = \langle \{r{=}(i \mod 2), y{=}i\}, \{y{=}i+1\} \rangle$,
  - $a_z^i = \langle \{z{=}i\}, \{z{=}i+1\} \rangle$,
  - $A_{r\rightarrow 0} = \bigcup_{i=1}^{2m-3} \{\langle \{r{=}1, z{=}i\}, \{r{=}0\} \rangle \mid i \text{ is odd}\}$, and
  - $A_{r\rightarrow 1}^i = \bigcup_{u_j=\theta \in C_i} \{\langle \{r{=}0, z{=}2i, x_j{=}\theta\}, \{r{=}1\} \rangle\}$.

Note that the largest k-dependence in Π is 2. As pointed out above, a solution for Π exists iff the value of r can be changed $2m-1$ times, and the value of r can be changed $2m-1$ times iff there exists an assignment that satisfies clauses $C_0, \ldots, C_{m-1}$. As the initial value of r is 0, $2m-1$ changes of r indicates that r must change from 0 to 1 m times. Each of these changes must be caused by actions that are drawn from the sets $A_{r\rightarrow 1}^i$ for different values of i, since each consecutive change to r depends on different values of z. Due to the construction of the set of actions $A_{r\rightarrow 1}^i$, an action from this set can be applied iff C_i is satisfied. Therefore plan existence implies that all C_i are satisfied. ∎

We now consider the 1-dependent case, first proving a lemma that leads to our tractability result for hourglasses:

Lemma 1 (Optimal plans for 1-dependent Hourglasses) *Given an hourglass-structured 1-dependent planning problem Π with $|\mathcal{D}(r)| = 2$, there exists an optimal plan for Π in which the actions changing the value of r have prevail conditions on at most two variables.*

Proof: Let π^* be an optimal plan for Π, and let π_r^* be the subsequence of π^* consisting only of actions in A_r. For each $\theta \in \mathcal{D}(r)$, let $a_\theta^* = \text{argmin}_{a \in \pi_r^*} \{cost(a) \mid \mathsf{eff}(a)[r] = \theta\}$, and let x and x' be the two variables on which the actions a_θ^* for $\theta \in \mathcal{D}(r)$ have prevail conditions. If $x \neq x'$, then it is possible to construct from π^* a new optimal plan π'^* that uses only these cheapest actions to change the value of r. The remaining actions that change the values of other parent variables or those of the child variables can be left unchanged. Since the actions we replace are no cheaper than a_θ^*, the result is also an optimal plan. Note that this also holds if one or both of the cheapest actions have no prevail conditions.

For the more complicated case in which $x = x'$, let

$$(a', \theta') = \underset{a \in \pi_r^*, \theta \in \mathcal{D}(r)}{\text{argmin}} \left\{ \begin{array}{l|l} cost(a) + & \mathsf{eff}(a)[r] \neq \theta \wedge \\ cost(a_\theta^*) & \mathsf{pre}(a)[x] \text{ is unspecified} \end{array} \right\}$$

In words, a' is an action not prevailed by x that together with $a_{\theta'}^*$ gives the lowest summed cost for two actions changing r from one value to another and back, at least one of which is not prevailed by x. If such an action does not exist, then π_r^* complies with the above property. We now show how to

obtain an optimal plan for Π in which all of the r-changing actions are either prevailed by x or are occurrences of a'. Let a_1, a_2 denote two consecutive actions in π_r^* such that at least one of a_1, a_2 is not prevailed by x, and $\{a_1, a_2\} \neq \{a', a^*_{\theta'}\}$. If no such pair of consecutive actions exists, then the condition described above is met. Otherwise, we construct a new sequence $\pi_r'^*$ by inserting in π_r^* immediately after the first occurence of $a^*_{\theta'}$ the two actions $a', a^*_{\theta'}$, and removing the two actions a_1, a_2. As noted earlier, the summed cost of a' and $a^*_{\theta'}$ is minimal among two r-changing actions at least one of which is not prevailed by x, and $\pi_r'^*$ is therefore no more expensive than π_r^*. Since the value that prevails a' and the sequence of distinct values of x that prevail actions in $\pi_r'^*$ are achieved by π_r^*, a new plan can be constructed by scheduling the actions achieving these values appropriately with respect to the actions in $\pi_r'^*$. As above, actions affecting other parent variables and child variables can be left untouched. The result is an optimal plan π'^* that complies with the above property. ∎

Lemma 1 allows us to concentrate on optimal plans of a certain structure, and therefore solve this type of hourglass problem optimally in polynomial time:

Theorem 8 *Optimal planning for* 1*-dependent hourglasses with center variable domain size* $|\mathcal{D}(r)| = 2$ *is in* P.

Proof: For each subset V' of size 2 of the parents $\mathsf{pred}(r)$ we create a planning problem Π' by removing from Π all r-changing actions that have preconditions on the variables in $\mathsf{pred}(r) \setminus V'$. From lemma 1 we have that an optimal plan for one such problem Π' is also an optimal plan for our original problem Π. Since Π' consists of the set of singleton variables $\mathsf{pred}(r) \setminus V'$, each of which can be solved in polynomial time, along with the rest of the problem which is a semifork with a hat of size 2, which can also be solved in polynomial time (Theorem 3), Π' can also be solved optimally in polynomial time. Since the number of Π' problems that must be considered to find an optimal plan for Π is polynomial, optimal planning for Π is in P. ∎

Finally, note that the planning problem in the proof of Theorem 6 is 1-dependent, as the indegree of each state variable is bounded by 1. Even satisficing planning for 1-dependent hourglasses with $|\mathcal{D}(r)| \geq 3$ is therefore NP-hard, completing our complexity map of the hourglass fragment.

Practice

Our tractability results for cost-optimal planning suggest that implicit abstraction heuristics can be made more informative. A semifork with a single hat variable, for example, can naturally represent fuel constraints for a mobile in transportation domains (Helmert 2008). However, there are a number of issues which must be attended to before the semifork and hourglass patterns can be employed in the framework of structural pattern database heuristics. Given a planning task Π over the variables V and a variable $v \in V$, the first issue is how to select a semifork or hourglass centered at v. For a constant k bounding the size of the hat and a set of variables $V' \subseteq (V \setminus \{v\})$ of size $\leq k$, a semifork with hat V' can be constructed by dropping all outgoing edges from $L = V \setminus (V' \cup \{v\})$ and all edges from $V \setminus \{v\}$ to L, leaving only edges from v to L and among $V' \cup \{v\}$. As the number of such sets V' is polynomial in k, all possible such semiforks could be accounted for. Hourglasses are more problematic, as when there exists $v' \in V$ such that edges (v, v') and (v', v) are both in $CG(\Pi)$, there is a choice of whether to use v' as a child or a parent. The second issue is how to abstract the problem to have G as its causal graph, modifying the set of actions to be consistent with its edges. For hourglasses, the previously defined acyclic causal-graph decomposition (Katz and Domshlak 2010) can be used, but must be adapted to account for possible cycles in semiforks. The last issue is that the chosen set of abstractions must be efficiently solvable in the states encountered during search. For this, most of the calculations can be performed prior to search and cached. The choice of the values to be precalculated and stored remains a subject of research.

$\lvert\mathcal{D}(r)\rvert$	2	3	$O(1)$	$\Theta(n)$
F	P/—	NPC/—	—/P	—/NPC
SF	P/—	—/NPC		
$H_{\lvert Ch\rvert=O(1)}$	P/—			—/NPC
$H(1)$	P/—	—/NPC		
$H(2)$	—/NPC			

Figure 5: Complexity of cost-optimal/satisficing planning for **F**orks, **S**emi**F**orks with constant bound on hat size, and **H**ourglasses, with k-dependence in parentheses. "—" and empty columns indicate that the complexity is implied by other results. Results implied by previous work are shaded.

Conclusions

We have extended the analysis of the complexity of planning problems described in terms of the structure of the causal graph, k-dependence, and the domain sizes of variables (Figure 5). We have closed some gaps left open in previous work, showing that optimal planning for fork causal graphs with root variable domain size ≥ 3 is NP-complete, and that satisficing planning is in P for arbitrary constant sized domains. We have introduced new causal graph fragments, called the semifork and hourglass, that generalize the previously known fork and inverted fork structures. Optimal planning for semiforks with center variable domain size 2 and a constant bound on the number of variables in the hat turns out to be in P, as does optimal planning for hourglasses with binary center variable domain and k-dependence 1. Relaxing the bound on domain size in either case results in a problem that is NP-complete even for satisficing planning, and the same is true of relaxing the bound on k-dependence for hourglasses. A number of questions must be addressed before these patterns can be used in the framework of structural pattern database heuristics.

Acknowledgments. Work performed while Michael Katz was employed by INRIA, Nancy, France, supported by the

French National Research Agency (ANR), project ANR-10-CEXC-003-01.

References

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

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

Brafman, R. I., and Domshlak, C. 2006. Factored planning: How, when, and when not. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI)*, 809–814.

Chen, H., and Giménez, O. 2008. Causal graphs and structurally restricted planning. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS)*, 36–43.

Domshlak, C., and Dinitz, Y. 2001. Multi-agent off-line coordination: Structure and complexity. In *Proceedings of Sixth European Conference on Planning (ECP)*, 277–288.

Giménez, O., and Jonsson, A. 2008. The complexity of planning problems with simple causal graphs. *Journal of Artificial Intelligence Research* 31:319–351.

Giménez, O., and Jonsson, A. 2009. The influence of k-dependence on the complexity of planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 138–145.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 162–169.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS)*, 161–170.

Helmert, M. 2008. *Understanding Planning Tasks: Domain Complexity and Heuristic Decomposition*, volume 4929 of *Lecture Notes in Computer Science*. Springer.

Katz, M., and Domshlak, C. 2008. New islands of tractability of cost-optimal planning. *Journal of Artificial Intelligence Research* 32:203–288.

Katz, M., and Domshlak, C. 2010. Implicit abstraction heuristics. *Journal of Artificial Intelligence Research* 39:51 – 126.

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