29th International Conference on
Automated Planning and Scheduling

July 10–15, 2019, Berkeley, CA, USA

HSDIP 2019

Proceedings of the 11th Workshop on

Heuristics and Search for Domain-independent Planning (HSDIP)

Edited by:

Guillem Francès, Florian Geißer, Daniel Gnad, Patrik Haslum,
Florian Pommerening, Miquel Ramirez, Jendrik Seipp, and
Silvan Sievers

Organization

Guillem Francès
University of Basel, Switzerland

Florian Geißer
Australian National University, Australia

Daniel Gnad
Saarland University, Germany

Patrik Haslum
Australian National University, Australia

Florian Pommerening
University of Basel, Switzerland

Miquel Ramirez
University of Melbourne, Australia

Jendrik Seipp
University of Basel, Switzerland

Silvan Sievers
University of Basel, Switzerland

Foreword

Planning as heuristic search remains among the dominating approaches to many variations of domain-independent planning, including classical planning, temporal planning, planning under uncertainty and adversarial planning, for nearly two decades. The research on both heuristics and search techniqes is thriving, now more than ever, as evidenced by both the quality and the quantity of submissions on the topic to major AI conferences and workshops.

This workshop seeks to understand the underlying principles of current heuristics and search methods, their limitations, ways for overcoming those limitations, as well as the synergy between heuristics and search. To this end, this workshop intends to offer a discussion forum and a unique opportunity to showcase new and emerging ideas to leading researchers in the area. Past workshops have featured novel methods that have grown and formed indispensable lines of research.

This year is the 11th edition of the workshop series on Heuristics for Domain-independent Planning (HDIP), which was first held in 2007. HDIP was subsequently held in 2009 and 2011. With the fourth workshop in 2012, the organizers sought to recognize the role of search algorithms by acknowledging search in the name of the workshop, renaming it to the workshop on Heuristics and Search for Domain-independent Planning (HSDIP). The workshop continued flourishing under the new name and has become an annual event at ICAPS.

Guillem Francès, Florian Geißer, Daniel Gnad, Patrik Haslum,
Florian Pommerening, Miquel Ramirez, Jendrik Seipp, and Silvan Sievers
July 2019

Contents

Reshaping Diverse Planning: Let There Be Light!

Michael Katz and **Shirin Sohrabi**
IBM T.J. Watson Research Center
1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA

Abstract

The need for multiple plans to a planning problem has been established by various applications. In some, solution quality has the predominant role, while in others diversity is the key factor. Most recent work takes both plan quality and solution diversity into account under the generic umbrella of *diverse planning*. There is no common agreement, however, on a collection of computational problems that fall under that generic umbrella. This in particular might lead to a comparison between planners that have different solution guarantees or optimization criteria in mind. In this work we revisit diverse planning literature in search of such a collection of computational problems, classifying the existing planners to these problems. We formally define a taxonomy of computational problems with respect to both plan quality and solution diversity, extending the existing work. We propose a novel approach to diverse planning, exploiting existing classical planners via planning task reformulation and choosing a subset of plans of required size in post-processing. Based on that, we present planners for two computational problems, that most existing planners solve. Our experiments show that the proposed approach significantly improves over the best performing existing planners in terms of coverage, the overall solution quality, and the overall diversity according to various diversity metrics.

1 Introduction

Many applications of planning require generating multiple plans rather than one. Some examples include malware detection (Boddy et al. 2005), automated analysis of streaming data (Riabov et al. 2015), and risk management (Sohrabi et al. 2018). Planners that produce multiple plans were also found useful in the context of re-planning and plan monitoring (Fox et al. 2006), user preferences (Myers and Lee 1999; Nguyen et al. 2012), as well as the engine for plan recognition and its related applications (Sohrabi, Riabov, and Udrea 2016). All these applications justify the need for finding a *diverse* set of plans while keeping quality in mind.

Many diverse planners were developed over the last decade, each one focused on addressing a particular diversity metric. For example, while DLAMA focuses on finding a set of plans by considering a landmark-based diversity measure (Bryce 2014), LPG-d and DIV focus on finding a set of plans with a particular minimum action distance (Nguyen et al. 2012; Coman and Muñoz-Avila 2011). Goldman and Kuter (2015) propose a diversity metric based on information retrieval literature. Roberts, Howe, and Ray (2014) suggest another diversity metric, introducing several planners, such as itA^* and MQA, which, in addition to the diversity metrics, consider plan quality. Recently, Vadlamudi and Kambhampati (2016) suggested "cost-sensitive" diverse planners, first finding all cost sensitive plans and then finding a diverse set of plans among these. Top-k planners (e.g., Katz et al. 2018b) or top-quality planners (Katz, Sohrabi, and Udrea 2019b) can also be viewed as diverse planners, which find a set of plans purely addressing the quality metric.

Despite the large number of existing tools and diversity metrics, there is no adopted collection of computational problems in diverse planning, making the comparison of different approaches challenging. Further, mixing quality and diversity creates an additional challenge for comparing various planners, especially if they have different optimality guarantees. However, even for the same computational problem, planner comparison can be challenging. Every planner can have a different implementation of the same diversity metric, and many planners produce a collection of plans without specifying the metric used, or the solution value under that metric. To the best of our knowledge, there exists no external validation tool for a collection of plans, producing the solution value under a given diversity metric. Additionally, most of the diverse planning approaches compute the set of plans by repeatedly solving the same task. To obtain a different behavior, planner's heuristic guidance is modified to account for already found plans, with a specific focus on a particular metric. This requires (a) having an intimate familiarity with the way a particular planner works, and (b) creating a separate modification for each metric. However, the outcome is not always as intended. Tweaking the heuristic function does not necessarily result in a different plan and planners have to discard many equal plans and repeat unnecessary iterations.

In this work, we address the computational problems in diverse planning as well as the diverse planner construction paradigm. Similarly to the separation in classical planning, we distinguish between optimal, bounded, and satisficing diverse planning and map the existing planners to their respective categories. We propose a new quality metric for a set of plans, measuring how close the plans are to the best

subset of all known plans. We create an external validation tool for the metrics considered in this paper, allowing us to compute the diversity values of the solutions produced by existing planners. We introduce an alternative planner construction paradigm, a diverse planning algorithm that instead of modifying a planner, modifies a planning task. Following the ideas of Katz et al. (2018b), we suggest reformulating the planning task after each iteration, forbidding sets of plans. Next, we post-process the found plans to derive a subset of plans of the required size, according to the given metric. Our approach, Forbid Iterative (FI), is not restricted to any planner and can exploit the recent advances in classical planning. To demonstrate this advantage, we experiment with one of the recent best-performing approaches to agile planning, heuristic novelty of the red-black planning heuristic (Katz et al. 2017; Katz, Hoffmann, and Domshlak 2013; Domshlak, Hoffmann, and Katz 2015), a core component for several participants of the recent International Planning Competition (IPC) 2018 (Katz et al. 2018a; Katz 2018). Based on this approach, we create planners for two of the introduced computational problems. We show that the same approach outperforms the dedicated planners built for specific metrics on these metrics and on their linear combinations, for both computational problems.

2 Preliminaries and Related Work

A SAS$^+$ *planning task* (Bäckström and Nebel 1995) is given by a tuple $\langle \mathcal{V}, \mathcal{A}, s_0, s_* \rangle$, where $\mathcal{V}$ is a set of *state variables*, $\mathcal{A}$ is a finite set of *actions*. Each state variable $v \in \mathcal{V}$ has a finite domain $dom(v)$. A pair $\langle v, \vartheta \rangle$ with $v \in \mathcal{V}$ and $\vartheta \in dom(v)$ is called a *fact*. A (partial) assignment to $\mathcal{V}$ is called a *(partial) state*. Often it is convenient to view partial state p as a set of facts with $\langle v, \vartheta \rangle \in p$ if and only if $p[v] = \vartheta$. Partial state p is *consistent* with state s if $p \subseteq s$. We denote the set of states of a planning task by $\mathcal{S}$. s_0 is the *initial state*, and the partial state s_* is the *goal*. Each *action* a is a pair $\langle pre(a), eff(a) \rangle$ of partial states called *preconditions* and *effects*. An *action cost* is a mapping $C : \mathcal{A} \to \mathbb{R}^{0+}$. An action a is applicable in a state $s \in \mathcal{S}$ if and only if $pre(a)$ is consistent with s. Applying a changes the value of v to $eff(a)[v]$, if defined. The resulting state is denoted by $s[\![a]\!]$. An action sequence $\pi = \langle a_1, \ldots, a_k \rangle$ is applicable in s if there exist states $s_0, \cdots, s_k$ such that (i) $s_0 = s$, and (ii) for each $1 \leq i \leq k$, a_i is applicable in s_{i-1} and $s_i = s_{i-1}[\![a_i]\!]$. We denote the state s_k by $s[\![\pi]\!]$. π is a plan iff π is applicable in s_0 and s_* is consistent with $s_0[\![\pi]\!]$. We denote by $\mathcal{P}(\Pi)$ (or just $\mathcal{P}$ when the task is clear from the context) the set of all plans of Π. The cost of a plan π, denoted by $C(\pi)$ is the summed cost of the actions in the plan.

The distance between two plans π, π' is defined as $\delta(\pi, \pi') = 1 - \text{sim}(\pi, \pi')$, where the similarity measure sim is between 0 (two plans are unrelated) and 1 (equivalent). The diversity of a set of plans, $D(P)$, $P \subseteq \mathcal{P}$ is then defined as some aggregation (e.g., min or average) of the pairwise distance within the set P. While some domain-dependent similarity measures exist (e.g., Myers and Lee 1999; Coman and Muñoz-Avila 2011), recent research has focused on domain-independent measures, comparing plans based on their actions, states, causal links, or landmarks (Nguyen et al. 2012; Bryce 2014).

Stability similarity (inverse of the plan distance (Fox et al. 2006; Coman and Muñoz-Avila 2011)) measures the ratio of the number of actions that appear on both plans to the total number of actions on these plans, referring to plans as action sets, ignoring repetitions. Given two plans π, π', it is defined as $\text{sim}_{\text{stability}}(\pi, \pi') = |A(\pi) \cap A(\pi')| / |A(\pi) \cup A(\pi')|$, where $A(\pi)$ is the set of actions in π. Uniqueness similarity (Roberts, Howe, and Ray 2014) is another measure that considers plans as action sets. It measures whether two plans are permutations of each other, or one plan is a partial plan (subset) of the other plan. State similarity measures similarity between two plans based on representing the plans as a sequence of states, where each state is a set of predicates. While there are multiple ways to define state similarity, we adapt the following definition from (Nguyen et al. 2012), modifying it based on use of similarity rather than distance between plans. Let $(s_0, s_1, \ldots, s_k)$ and $(s_0, s'_1, \ldots, s'_{k'})$ be the sequences of states traversed by the plans π and π', respectively. Let $\Delta(s, s') = |s \cap s'| / |s \cup s'|$ be the similarity between two states. Assuming $k' \leq k$, the state similarity measure is defined as follows: $\text{sim}_{\text{state}}(\pi, \pi') = \sum_{i=1}^{k'} \Delta(s_i, s'_i)$ k. Note, each state $s'_{k'+1}, ..., s_k$ is considered to not contribute to the similarity measure (i.e., zero is considered). The combination of the state and uniqueness measures address some of the major weaknesses of the stability measure raised by recent research (Goldman and Kuter 2015). Thus, since our focus in this work is not on metrics, we omit the description of the landmark-based distance (Bryce 2014).

While there seems to be no widely adopted definitions of diverse planning problems, previous work has introduced some such definitions. In these definitions, d is a threshold on the distance and c is a threshold of the cost of the plans. The variant introduced by Nguyen et al. (2012) requires the distance between every pair of plans in the solution to be of bounded diversity. Formally, the search problem is depicted as follows:

$$\begin{aligned} &\text{dDISTANTkSET : find } P \text{ with } P \subseteq \mathcal{P}, \\ &|P| = k, \min_{\pi, \pi' \in P} \delta(\pi, \pi') \geq d. \end{aligned} \tag{1}$$

Another variant, by Vadlamudi and Kambhampati (2016) extends the previous search problem by requiring each individual plan in the solution to be of bounded quality. Formally:

$$\begin{aligned} &\text{cCOSTdDISTANTkSET : find } P \text{ with } P \subseteq \mathcal{P}, \\ &|P| = k, \min_{\pi, \pi' \in P} \delta(\pi, \pi') \geq d, C(\pi) \leq c \ \forall \pi \in P. \end{aligned} \tag{2}$$

While Eq. 2 considers plan costs, Eq. 1 only considers the distance between plan pairs. Note that both definitions require finding k distinct plans.

We denote the diversity of a set of plans P, computed as an average over the pairwise dissimilarity of the set P, under the similarity measures of stability, uniqueness, and state by D_a, D_u, and D_s, respectively, dropping P for readability. Also, D_{ma} denotes the diversity metric computed as minimum over the pairwise stability dissimilarity.

3 Quality Metric

While most work in diverse planning focused on the diversity metrics, not much was done for quality metrics. One possible quality metric is the summed cost of plans $Q = \sum_{\pi \in P} C(\pi)$. In order to normalize its value, it is possible, as with the International Planning Competition (IPC) quality metric for individual plans, to divide the best known solution value by the value of the given planner. One downside of such a metric is that a single plan's quality can have a large effect on the overall quality. For example, the quality of a set of plans, where all plans are optimal except for one, of a much higher cost, may get a quality score worse than a set where all plans are not optimal. Thus, we suggest a quality metric that will allocate a score to each plan in the set, aggregating these scores into a single score.

Given n diverse planners, let $P = P_1 \cup P_2 ... \cup P_n$ be the set of all plans found by these planners. Let $\pi_1, \dots \pi_k$ be k plans with the lowest cost, ordered by their cost from smallest to largest and let $c_i = C(\pi_i)$. For a planner j, the quality of the solution P_j is measured relatively to the best known k plan costs $c_1, \dots, c_k$ as follows. Let $\pi^j_1, \dots \pi^j_k$ be an ordering of plans in P_j according to their costs and let $c^j_i = C(\pi^j_i)$. The quality metric is defined as follows.

$$Q(P_j) := \frac{1}{k} \times \sum_{i=1}^{k} \frac{c_i}{c^j_i}. \tag{3}$$

Note that $c^j_i \geq c_i$, since $P_j \subseteq P$, and thus π^j_i has at least $i - 1$ plans of no larger cost in P. Thus, each sum component is between 0 and 1, and thus the whole score is a value between 0 and 1. Further, a solution P_j will get the score 1 if and only if it consists of k cheapest plans found by any planner. In other words, if there exists no plan in $P \setminus P_j$ (found by any of the other planners) that is cheaper than a plan in P_j. The suggested metric is similar in spirit to the parsimony ratio (Roberts, Howe, and Ray 2014). The parsimony ratio is defined as $s(\pi_k, \pi_l) = |\pi_k|/|\pi_l|$, where for each π_l ($|\pi_l| = l$) we need to find an optimal plan, π_k ($|\pi_k| = k$), such that $\pi_k \subseteq \pi_l$, $k \leq l$. This can be challenging by itself, since it requires finding optimal plans. The parsimony ratio also only considers unit cost plans. Both these limitations do not exist in our suggested metric: it can handle general costs and the computation is relative to the set of known plans.

4 Diverse Planning Revisited

In this section, we define a collection of computational problems in diverse planning for two optimization criteria, quality and diversity. Following previous definitions, depicted in Eqs. 1 and 2, we define a solution to a diverse planning problem as a set of plans of a required size. In contrast to previous definitions, in case there exist fewer plans than requested, the set of all plans is also considered to be a valid solution.

Definition 1 (Diverse planning solution) *Let Π be a planning task and $\mathcal{P}$ be the set of all plans for Π. Given a natural number k, $P \subseteq \mathcal{P}$ is a k-diverse planning solution if $|P| = k$ or $P = \mathcal{P}$ if $|\mathcal{P}| < k$.*

Restricting our attention to two optimization criteria, quality and diversity, let us introduce some terminology. We say that a solution is *quality-optimal* (*diversity-optimal*) if there exists no solution of better quality (diversity). In other words, given solution quality mapping Q (diversity mapping D), a solution P is quality-optimal (diversity-optimal) if for all solutions P' we have $Q(P') \leq Q(P)$ ($D(P') \leq D(P)$). Given a bound b, we say that a solution P is *quality-bounded* (*diversity-bounded*) if $Q(P) \geq b$ ($D(P) \geq b$).

For both quality and diversity, one could either strive to find optimal or bounded solutions, or impose no restriction on solution quality. Unfortunately, these two optimization criteria can interfere with each other. Thus, in what follows, we define various search and optimization problems.

4.1 Satisficing Diverse Planning

We start with imposing no restrictions. Thus, the *Satisficing Diverse Planning* problem can be defined as follows.

sat-k : Given k, find a k-diverse planning solution.

Note that the objective is to find any set of k plans without any restrictions on either quality or diversity. This is the category under which most diverse planners fall (e.g., Bryce 2014; Roberts, Howe, and Ray 2014). To compare planners in this category, it is sufficient to compare the quality and diversity of their solutions. Note, many of the satisficing diverse planners incorporate the distance measure into their search and focus on finding diverse plans with respect to that particular distance measure in mind. Hence, while they may perform well for one diversity metric, they may do poorly in another one.

4.2 Bounded Diverse Planning

Continuing now by restricting either quality or diversity by imposing a bound, we introduce a *Bounded Quality (Diversity) Diverse Planning*. We do that by restricting the set of feasible solutions.

Definition 2 (Diversity-bounded solution) *Let Π be a planning task, D be some diversity metric, b be some bound, and $\mathcal{P}$ be the set of all Π's plans. Given a natural number k, $P \subseteq \mathcal{P}$ is a b-diversity-bounded k-diverse planning solution if it is a k-diverse planning solution and $D(P) \geq b$.*

Definition 3 (Quality-bounded solution) *Let Π be a planning task, Q be some quality metric, c be some bound, and $\mathcal{P}$ be the set of all Π's plans. Given a natural number k, $P \subseteq \mathcal{P}$ is a c-quality-bounded k-diverse planning solution if it is a k-diverse planning solution and $Q(P) \geq c$.*

Given the definitions above, we can now define the following search problems:

bD-k : Given k and b, find a b-diversity-bounded k-diverse planning solution,

bQ-k : Given k and c, find a c-quality-bounded k-diverse planning solution.

Note that solutions for bQ-k can be obtained from solutions to the Top-quality planning problem (Katz, Sohrabi, and Udrea 2019b).

The search problem bD-k generalizes the definition in Eq. 1 by Nguyen et al. (2012), for a diversity score D_{ma} defined as the minimum over the pairwise stability dissimilarity. Note that this measure differs from D_a, that averages over the pairwise stability dissimilarity. For bounded diverse planning, D_{ma} dominates D_a in the sense that solutions to the diversity-bounded diverse planning under D_{ma} are necessarily solutions to the diversity-bounded diverse planning under D_a with the same bound, but not the other way around. The planner LPG-d implements the approach of Nguyen et al. (2012), for a variant of D_{ma}, where the stability similarity is computed over multisets, instead of sets. We denote this diversity metric by D_{mma}. Thus, LPG-d can be thought of as a diversity-bounded diverse planner for D_{mma} but not for any of the other metrics. Further, while LPG-d is a sound planner, it is not complete, since it can only add plans to the collection of previously found plans, and never reconsiders the decision to add a plan. Thus, in principle LPG-d might not be able to find a solution to the diversity-bounded diverse planning problem when a solution exists.

Restricting both quality and diversity results in an additional search problem, one we call *Bounded Quality and Diversity Diverse Planning*.

bQbD-k : Given $k, b,$ and c, find a c-quality-bounded and b-diversity-bounded k-diverse planning solution.

The search problem bQbD-k generalizes the definition in Eq. 2 by Vadlamudi and Kambhampati (2016), for diversity score that uses min as the aggregation method and quality score defined as a maximum over the individual plan costs. It is worth noting here that in all these definitions, as in classical planning, if the bound is super-optimal, the search problem is considered to be unsolvable.

4.3 Optimal Diverse Planning

Restricting now either the quality or diversity to be optimal, we define two optimization problems, *Optimal Quality (Diversity) Diverse Planning*.

optQ-k : Given k, find a quality-optimal k-diverse planning solution.

optD-k : Given k, find a diversity-optimal k-diverse planning solution.

Top-k planners (e.g., Riabov, Sohrabi, and Udrea 2014; Katz et al. 2018b) can be viewed as planners for optQ-k, optimizing the quality metric $Q = \sum_{\pi \in P} C(\pi)$. To the best of our knowledge, there are no existing planners for the optD-k optimization problem. In fact, it is not clear how to create such non-trivial planners, without the need to generate the set of all plans.

If we further restrict the other optimization function, this results in additional optimization problems. The first two

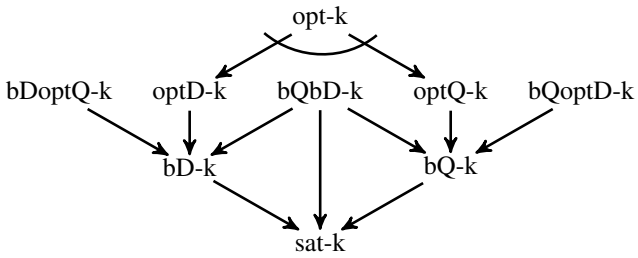


Figure 1: Hierarchy between the computational problems.

are *Optimal Quality Bounded Diversity Diverse Planning* and *Optimal Diversity Bounded Quality Diverse Planning*, as follows.

bDoptQ-k : Given k and b, find a quality-optimal among b-diversity-bounded k-diverse planning solutions.

bQoptD-k : Given k and c, find a diversity-optimal among c-quality-bounded k-diverse planning solutions.

Note that the solutions to the optimization problems bDoptQ-k and bQoptD-k are relative to the restricted set of solutions as in Definitions 2 and 3, respectively. This means that a solution to, e.g., bQoptD-k is not necessarily a solution to optD-k. One possible way to obtain solutions to the bQoptD-k optimization problem is by using a top-k planner to generate a set of all plans of bounded quality and then select an optimal subset of size k from the generated set according to some diversity metric.

We can further restrict a set of feasible solutions to quality-optimal (diversity-optimal) diverse planning solutions and choose the best according to the diversity (quality) metric among those. Instead, our last optimization problem we simply call *Optimal Diverse Planning*. The objective of optimal diverse planning is to find a solution that is *pareto-optimal*, that is for all solutions P' we have either $Q(P') \leq Q(P)$ and for all P'' with $Q(P) = Q(P'')$ we have $D(P'') \leq D(P)$ or $D(P') \leq D(P)$ and for all P'' with $D(P) = D(P'')$ we have $Q(P'') \leq Q(P)$. In words, optimal solutions are solutions on the pareto frontier of quality and diversity. We denote the optimization problem stated above by opt-k.

The hierarchy between the presented computational problems is depicted in Figure 1. Edges represent solution set inclusion, i.e., whether a solution for one problem is necessarily a solution for another, assuming a solution exists. For example, a pareto-optimal solution is a solution to either optD-k or optQ-k, but not necessarily to either bQoptD-k or bDoptQ-k, since the latter two optimize over the solutions that are of bounded quality and diversity, respectively. The diagram does not reflect the transitive inclusion, which, in this case, means that solutions to all problems are solutions to the satisficing diversity planning problem.

5 Satisficing Diverse Planning with a Satisficing Classical Planner

Previous work has focused on modifying existing planners, either heuristic search or local search based ones, to come up with plans that differ from previously found ones. These modified planners were then applied to the same planning task, over and over again. We suggest a different approach, using possibly the same planner, iteratively modifying the planning tasks to forbid plan sets (Katz et al. 2018b). Below, we list some of the benefits to such an approach: (1) it allows us to exploit state-of-the-art classical planners without the need to modify them, taking advantage of the progress in classical satisficing planning; (2) it removes the need for modifying the behaviour of the existing planners, allowing these planners to work as intended; (3) this allows us to take the selection of a subset of plans that is diverse according to a specific metric and postprocess them, thus also allowing us to define and use more sophisticated metrics.

5.1 Forbidding a Plan as a Multiset of Actions

Existing literature suggests one such task reformulation, forbidding exactly the given set of plans (Katz et al. 2018b). This was done in the context of top-k planning, where plans could not be safely discarded from consideration. In satisficing diverse planning, there is no such limitation. As a result, it is possible to forbid additional plans. One could envision a metric-dependent reformulation, forbidding also the plans that are similar according to the given metrics. With the stability metric in mind, we suggest a reformulation that ignores orders between actions in a plan and thus, also forbids all possible reorderings of a given plan. Below, we present the detailed description of such a reformulation.

Definition 4 *Let $\langle \mathcal{V}, A, s_0, s_* \rangle$ be a planning task and X be a multiset of actions. The task $\Pi^-_X = \langle \mathcal{V}', A', s'_0, s'_* \rangle$ is defined as follows.*

- $\mathcal{V}' = \mathcal{V} \cup \{\overline{v}\} \cup \{\overline{v}_o \mid o \in X\}$*, with $\overline{v}$ being a binary variable, and $dom(\overline{v}_o) = \{0, \ldots, m_o\}$, where m_o is the number of occurences of o in X,*
- $A' = \{o^e \mid o \in A \setminus X\} \cup \{o^r, o^d \mid o \in X\} \cup \bigcup_{i=1}^{m_o} \{o^f_i \mid o \in X\}$*, where*

  $o^e = \langle pre(o), eff(o) \cup \{\langle \overline{v}, 0 \rangle\} \rangle$,
  $o^r = \langle pre(o) \cup \{\langle \overline{v}, 0 \rangle\}, eff(o) \rangle$,
  $o^d = \langle pre(o) \cup \{\langle \overline{v}, 1 \rangle, \langle \overline{v}_o, m_o \rangle\}, eff(o) \cup \{\langle \overline{v}, 0 \rangle\} \rangle$,
  $o^f_i = \langle pre(o) \cup \{\langle \overline{v}, 1 \rangle, \langle \overline{v}_o, i\text{-}1 \rangle\}, eff(o) \cup \{\langle \overline{v}_o, i \rangle\} \rangle$,

  $C'(o^e) = C'(o^r) = C'(o^d) = C'(o^f) = C(o)$,
- $s'_0[v] = s_0[v]$ *for all* $v \in \mathcal{V}$, $s'_0[\overline{v}] = 1$, *and* $s'_0[\overline{v}_o] = 0$ *for all* $o \in X$, *and*
- $s'_*[v] = s_*[v]$ *for all* $v \in \mathcal{V}$ *s.t.* $s_*[v]$ *defined, and* $s'_*[\overline{v}] = 0$.

Let us explain the semantics of the reformulation in Definition 4. By X_π we denote the multiset of actions in a plan π. The variable $\overline{v}$ starts from the value 1 and switches to 0 when an action is applied that is not from the multiset $X = X_\pi$. Once a value 0 is reached indicating a deviation from plan π, it cannot be switched back to 1. Variables $\overline{v}_o$ encode the number of applications of the action o. The actions o^r and o^d are copies of the action o in X for the cases when π is already discarded from consideration (variable $\overline{v}$ has switched its value to 0) and for discarding π from consideration (switching $\overline{v}$ to 0), respectively. The latter happens if the action o was already applied as many times as it appears in X. o^f_i are copies of the action o in X, counting the number of applications of o, as long as the number is not higher than the number of times it appears in X. These actions are applicable only while the plan is still followed. As mentioned above, ignoring plan reorderings sits well with the stability metric, but also with the uniqueness metric. For the state metric, note that although different reorderings of the same plan produce different sequences of states, these sequences will mostly be quite similar. Thus, we believe that it is more beneficial to spend the time on finding additional plans that are "set"-different instead of finding additional reorderings of the found plans. Note that in principle we could do both, if time permits.

Algorithm 1 Iterative diverse planning scheme.

Input: Planning task Π, number of diverse plans k, number of total plans for search phase K, diversity metric D

```
P ← ∅
Π' ← Π
while |P| < K do
    π ← some solution to Π'
    P ← P ∪ {π' | π' is symmetric to π}
    X ← ⋃_{π∈P} X_π
    Π' ← Π^-_X according to Definition 4
end while
return choose k diverse plans from P, according to D
```

When a set of plans is available, obtained, e.g., by applying structural symmetries (Shleyfman et al. 2015), one option would be to reformulate via a series of reformulations as in Definition 4. Another option is to forbid possibly more than just that set of plans by exploiting Definition 4 for forbidding a multiset of actions that is a superset of all plans in the set. In our implementation, we decided to follow the latter approach, depicted in Algorithm 1. Each iteration starts from the original task and forbids all plans found so far. In the last step, the algorithm selects a diverse subset of plans out of the set of plans found so far. In what follows, we discuss how such a selection can be done.

5.2 Selecting a Diverse Subset of Plans

The idea of selecting a set of plans in a post-processing phase is not new. A basic filtering and then clustering was performed over the set of plans for a top-k planning problem (Sohrabi et al. 2016; 2018). These approaches, however, may become time consuming when metric computation is computationally expensive. Hence, in this work, we instead apply a simple greedy algorithm, with a negligible computational overhead. We first order the found plans by their cost. Then, going from the cheapest plans to the more expensive ones, we find a pair of plans with the largest diversity score.

Starting with the found pair of plans, we iteratively construct the set by greedily choosing the next plan to add to the set, maximizing the diversity of the resulting set at that iteration step. We stop once the set reaches the requested size k. We note that the quality of the solution obtained by such an algorithm may be considerably improved. However, as we see next, even such a naive algorithm produces quite encouraging results.

6 Diversity-Bounded Diverse Planning

As previously mentioned, LPG-d as described by Nguyen et al. (2012) is a sound diversity-bounded diverse planner, although not complete. Similarly, our suggested approach can be used to produce a sound diversity-bounded diverse planner by post-processing the obtained plans differently. In general, such a post-processing procedure should find a collection of plans that adhere to certain constraints and that often corresponds to solving an NP-hard computational problem. For D_{mma}, that corresponds to finding a clique of size at least k, for a graph over vertices that correspond to plans found during the search phase and edges that correspond to pairs of plans of stability dissimilarity of at least d. Such cliques can be found using, e.g., mixed-integer linear program tools. In what follows, we use binary variables, one for each graph vertex to encode whether the vertex is a part of the selected clique. For each pair of vertices that are not connected by an edge, at most one of these vertices can belong to a clique. Thus, we introduce a constraint stating that if there is no edge between two vertices, then the sum of the two corresponding binary variables cannot exceed 1. An additional constraint requires the sum of all binary variables to be greater or equal to k, the number of the requested plans. Thus, valid assignments to the binary variables correspond exactly to cliques of size at least k. As a result, any optimization criteria can be chosen. Here, we choose to minimize the size of the obtained clique, finding a clique of size exactly k. This is done by minimizing the sum of all variables. Note that, while it is not required by the diversity-bounded diversity-bounded diverse planning problem, one can optimize other criteria while keeping the same set of constraints, and choosing a clique, e.g., maximizing the sum of pairwise stability measures.

7 Experimental Evaluation

In order to evaluate the feasibility of our suggested approach for deriving diverse sets of plans according to various existing metrics, we have implemented our approach on top of the Fast Downward planning system (Helmert 2006). Our planners, ForbidIterative (FI) diverse planners are available as part of the collection of ForbidIterative planners (Katz, Sohrabi, and Udrea 2019a). Further, we implemented an external component, that given a set of plans and a metric returns the score of the set under that metric (Katz and Sohrabi 2019).

We compare our approach for satisficing diverse planning to existing satisficing diverse planners, namely DLAMA planner (Bryce 2014), DIV (Coman and Muñoz-Avila 2011), itA^*, RWS, MQAd, MQAs, MQAtd, and MQAts (Roberts, Howe, and Ray 2014), on state, stability, uniqueness, as well as a uniform linear combination over all subsets of these metrics, seven diversity metrics overall, shown in Table 1. Our diversity-bounded diverse planner is compared to the only existing diversity-bounded diverse planner LPG-d (Nguyen et al. 2012), on the D_{mma} metric, varying the diversity parameter d to obtain values 0.15, 0.25, and 0.5 (see Table 2). We also varied the value of k, the number of required plans, for $k \in \{5, 10, 100, 1000\}$. For completeness, we include a comparison to LPG-d viewed as a satisficing diverse planner. To compare to all selected existing planners, we restrict our benchmark set to STRIPS domains with uniform action costs from the International Planning Competitions (IPC). This results in 1276 tasks in 40 domains.

The experiments were performed on Intel(R) Xeon(R) CPU E7-8837 @2.67GHz machines, with time and memory limits of 30min and 2GB, respectively. Our suggested approach iteratively solves a planning task, finds a set of plans, and creates a new task that forbids a superset of the plans found so far. Considering plans as multisets, ignoring the order between the actions, this superset is defined as the union of all plans found so far. Thus, we forbid reorderings of found plans, but also, possibly additional plans, corresponding to a union of multiple found plans. We are restricting the number of found plans to 1000. For solving the (original and reformulated) planning tasks, we use an existing state-of-the-art agile planner. The planner that was chosen is *MERWIN* (Katz et al. 2018a). It performs a greedy best-first search (GBFS), alternating between four queues, novelty of the red-black heuristic, landmark count, preferred operators from the red-black heuristic, and preferred operators from the landmark count heuristic. The configuration has shown an exceptionally good performance on the IPC domains in our benchmark set (Katz et al. 2017). Note that while we report only results for MERWIN, we have also experimented with LAMA (Richter and Westphal 2010). The results were similar, therefore we report here only the results for MERWIN. A minor restriction in our choice of an external planner is the ability to work directly on SAS$^+$ representation, since our task reformulation is performed directly on SAS$^+$ and results in a SAS$^+$ task. This restriction is indeed somewhat minor, since most state-of-the-art planners do work on the grounded SAS$^+$ representation. In some cases, however, an adaptation might be required, since our implementation uses the input format of the Fast Downward planning system (Helmert 2006).

The solution to the computational problem of interest is chosen in the post-processing step from the found plans. Focusing first on satisficing diverse planning, if the desired number of plans k is lower, we then greedily[1] choose a subset of size k according to the given diversity metrics, as described in Section 5.2. Note that this can result in different subsets of plans chosen for different metrics. The algorithm is implemented as part of the external component (Katz and Sohrabi 2019). Each technique gets a score between 0 and 1 for each task and each metric, as described in previous

[1] We have experimented with exact techniques, based on mixed-integer linear programs, but found them to be prohibitively slow.

		FI	DIV	DLAMA	itA*	LPG-0.15	LPG-0.25	LPG-0.5	MQAd	MQAs	MQAtd	MQAts	RWS
k=5	coverage	**1143**	95	178	611	705	701	680	277	499	2	615	51
	Q_c	**1095.66**	84.69	127.26	539.13	527.10	526.46	488.91	190.37	447.37	1.80	533.03	39.25
	D_a	**736.88**	33.65	123.07	271.77	402.72	412.39	455.86	213.83	277.70	0.40	322.17	30.62
	D_s	**585.34**	45.01	96.35	200.58	321.62	322.02	336.41	144.86	143.73	1.09	229.62	25.32
	D_u	**1093.70**	53.10	176.00	527.70	688.10	689.10	671.90	275.20	486.70	0.60	539.40	41.20
	D_s D_a	**640.46**	39.33	108.60	236.17	362.13	367.17	396.03	179.34	210.71	0.74	275.90	27.97
	D_s D_u	**837.18**	49.06	136.19	364.14	504.81	505.55	504.14	210.03	315.21	0.85	384.51	33.26
	D_u D_a	**915.87**	43.37	149.50	399.74	545.39	550.74	563.87	244.51	382.20	0.50	430.79	35.91
	D_a D_u D_s	**791.81**	43.92	131.11	333.35	470.75	474.45	487.97	211.30	302.71	0.70	363.73	32.38
k=10	coverage	**1113**	1	133	422	661	652	610	168	398	0	430	31
	Q_c	**1060.08**	0.93	92.43	376.64	508.15	500.38	433.88	106.96	361.55	0.00	363.00	23.68
	D_a	**681.08**	0.48	88.79	191.66	384.80	394.07	418.92	136.97	222.69	0.00	219.30	21.38
	D_s	**534.93**	0.52	71.32	164.53	300.97	302.22	307.07	93.88	114.76	0.00	175.29	15.84
	D_u	**1054.53**	1.00	132.62	353.76	648.07	645.27	608.53	167.60	394.91	0.00	365.40	28.91
	D_s D_a	**590.98**	0.50	79.38	178.10	342.86	348.10	362.80	115.43	168.73	0.00	197.29	18.61
	D_s D_u	**792.08**	0.76	101.92	259.14	474.51	473.71	457.80	130.74	254.84	0.00	270.34	22.37
	D_u D_a	**868.10**	0.74	110.68	272.71	516.42	519.66	513.73	152.28	308.80	0.00	292.35	25.15
	D_a D_u D_s	**745.40**	0.67	97.09	236.65	444.59	447.13	444.71	132.82	244.12	0.00	253.33	22.04
k=100	coverage	**909**	0	11	37	550	535	433	32	170	0	78	15
	Q_c	**849.21**	0.00	7.28	31.80	450.36	431.24	296.07	17.41	165.39	0.00	66.48	11.61
	D_a	**492.55**	0.00	6.89	22.12	339.17	337.18	319.36	27.02	100.35	0.00	51.24	11.53
	D_s	**404.63**	0.00	5.78	17.90	262.98	258.64	227.79	18.96	57.49	0.00	34.09	7.70
	D_u	**834.18**	0.00	11.00	32.70	548.63	534.11	432.95	31.99	169.99	0.00	76.74	14.95
	D_s D_a	**438.70**	0.00	6.29	20.01	301.04	297.85	273.43	22.99	78.92	0.00	42.67	9.61
	D_s D_u	**617.02**	0.00	8.39	25.30	405.77	396.35	330.36	25.48	113.74	0.00	55.42	11.33
	D_u D_a	**661.18**	0.00	8.94	27.41	443.91	435.63	376.15	29.51	135.17	0.00	63.99	13.24
	D_a D_u D_s	**569.55**	0.00	7.86	24.24	383.55	376.58	326.60	25.99	109.28	0.00	54.02	11.39
k=1000	coverage	**552**	0	0	0	406	363	234	0	0	0	0	7
	Q_c	**543.14**	0.00	0.00	0.00	361.52	313.51	170.68	0.00	0.00	0.00	0.00	5.96
	D_a	**263.58**	0.00	0.00	0.00	244.58	224.01	174.19	0.00	0.00	0.00	0.00	5.38
	D_s	**206.54**	0.00	0.00	0.00	194.19	173.49	118.10	0.00	0.00	0.00	0.00	3.66
	D_u	**490.44**	0.00	0.00	0.00	405.93	362.92	234.00	0.00	0.00	0.00	0.00	7.00
	D_s D_a	**233.82**	0.00	0.00	0.00	219.35	198.71	146.10	0.00	0.00	0.00	0.00	4.52
	D_s D_u	**348.47**	0.00	0.00	0.00	300.05	268.20	176.05	0.00	0.00	0.00	0.00	5.33
	D_u D_a	**375.79**	0.00	0.00	0.00	325.25	293.45	204.11	0.00	0.00	0.00	0.00	6.20
	D_a D_u D_s	**318.92**	0.00	0.00	0.00	281.54	253.44	175.40	0.00	0.00	0.00	0.00	5.35

Table 1: Overall summed scores for various metrics, for k=5, 10, 100, and 1000. D_a stands for *stability*, D_s for *state*, and D_u for *uniqueness* diversity metrics. Rows that correspond to a linear combination of diversity metrics are marked with all combined metrics. Q_c stands for *cost* quality metric. Best results are highlighted in bold.

sections. If not enough unique plans were found by some planner on a task, the planner gets the score of 0 for that task. Table 1 depicts the summed scores for all planners on all metrics, for various values of k, from 5 to 1000. First, note that our approach excels on all metrics, for both diversity and quality. This is due in part to an increased coverage, by 62% for $k = 5$, 68% for $k = 10$, 65% for $k = 100$, and 36% for $k = 1000$. However, as we later see, that is not the only source of improved performance. For the quality metric, we improve by over 100% for the smaller values of $k = 5$ and 10, by 88% for $k = 100$, and by 50% for $k = 1000$. For various diversity metrics, the improvement is between 59% and 74% for $k = 5$ and 10, and between 45% and 54% for $k = 100$. For $k = 1000$, the improvement is much more modest: 6% for the *state* metric, 8% for the *stability* metric, and 21% for the *uniqueness* metric. Note that while for smaller k values there are several techniques that are somewhat comparable in their performance to ours, larger k values seem to be challenging for most techniques. The only exception is *LPG-d*, which performs rather well even for large k values. In fact, despite solving a different computational problem, *LPG-d* is the strongest competitor to our approach for all tested values of k.

In order to go beyond the aggregated results, Figure 2 shows the comparison between our technique and *LPG-d* with $d = 0.5$, the best performing contestant for $k = 5$. The plots show two diversity metrics, *stability* and *state*. Each task corresponds to a single point, with coordinates representing the metric value. All points above the diagonal are in favor of *LPG-d*, and below the diagonal are in favor of our technique. The points on the axes correspond to tasks that either were solved by one technique but not the other or the score obtained by one of the techniques was 0. For the metric stability in Figure 2(a), there are 884 tasks below the diagonal, 490 of these tasks are on the x axis. There are 286 tasks above the diagonal, 41 of these tasks are on the y axis. For the metric state in Figure 2(b), there are 938 tasks below the diagonal, with 495 tasks on x axis and 237

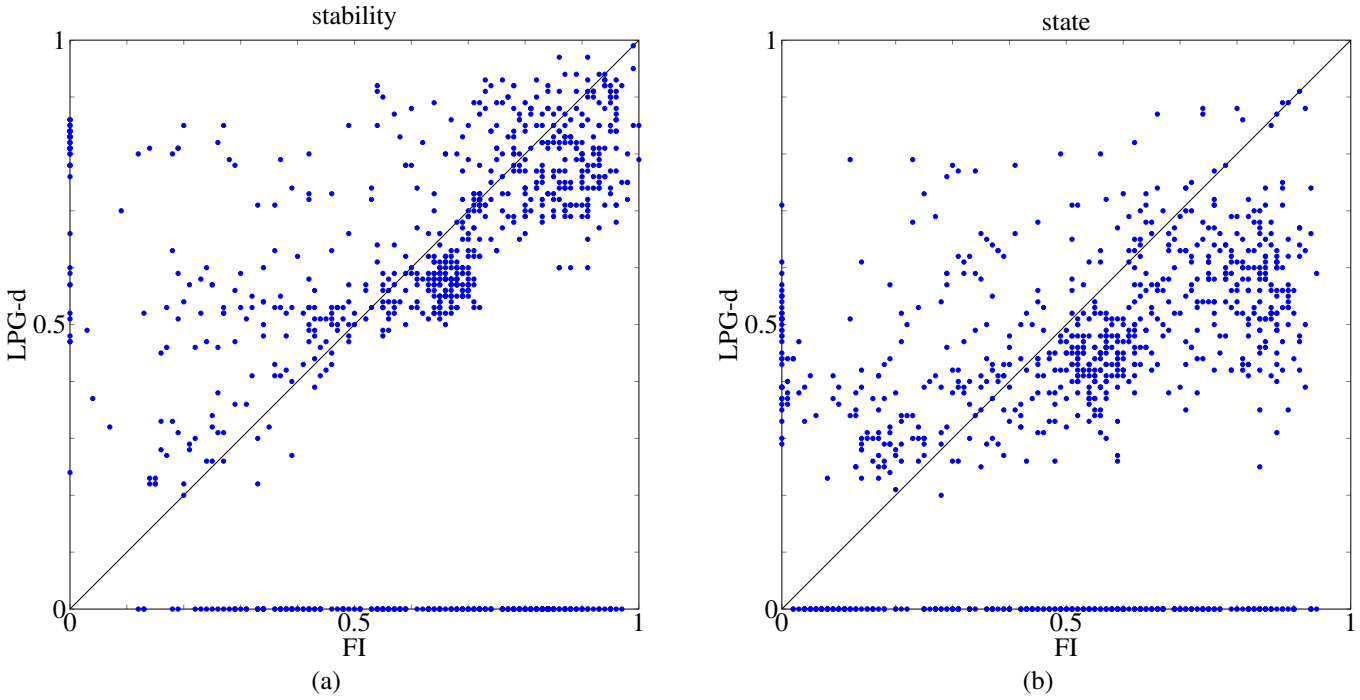


Figure 2: Comparison of our technique (FI) to the LPG-d planner with $d = 0.5$ on (a) D_a and (b) D_s metrics for $k = 5$.

	0.15			0.25			0.5		
k	**bFI**	**LPG-d**	Dom	**bFI**	**LPG-d**	Dom	**bFI**	**LPG-d**	Dom
5	**1011**	675	28/6	**974**	671	25/8	**890**	652	24/10
10	**946**	632	26/7	**912**	623	27/9	**771**	586	25/10
100	**569**	532	17/13	454	**517**	15/15	213	**433**	8/16
1000	152	**396**	7/17	80	**359**	3/18	3	**234**	1/15

Table 2: Comparison of bounded-diversity score (total number of solved tasks) for k=5, 10, 100, and 1000 for the *stability* metric (D_{mma}). Best results are bolded. Dom shows # of domains with superior performance for bFI/LPG-d.

tasks above the diagonal, with 39 tasks on y axis. Observe that most of the tasks are not near the diagonal, and thus these techniques are rather complementary. We note that for *FI*, the score was computed with a greedy algorithm. Exact solutions, although slower, might have got a better score.

Moving now to diversity-bounded diverse planning, we increased the bound on the number of plans found in the first phase to 2000, to give the planner some choice for $k = 1000$. The solution here is obtained by solving the binary linear program, as described in Section 4.2 with the CPLEX solver in its default configuration. The implementation is available as part of the external component (Katz and Sohrabi 2019). While in general these programs have up to 2K binary variables and up to 4M constraints, we observe that the run time of the solver is rarely above 10 seconds, with the peak reaching 47 seconds. If binary linear program was solved by the solver (feasible solution found), the planner gets 1, and otherwise (infeasible) 0. We post-process the set of plans from both our approach and LPG-d in the same way. Table 2 shows the overall summed scores over all instances, as well as the number of domains where each approach exhibits superior performance. As a reminder, our approach chooses k plans out of the found plans with D_{mma} above the given threshold. Thus, our approach has a clear disadvantage when there is little or no choice, as in the case of the largest k values in our experiment. For smaller k values ($k = 5, 10$), there is a clear advantage to our approach, for all tested bounds on D_{mma}.

8 Summary and Future Work

We have presented various diverse planning computational problems and classified the existing diverse planners with their respective problems. Key contributions of this paper include: (1) characterization of optimal, bounded, and satisficing diverse planning problem; (2) introducing an external validation component for diverse planning; (3) addressing the satisficing and bounded-diversity diverse planning problems by iteratively solving a modified planning task using existing classical planners, escaping the need to adapt a planner to each new diversity metric. We have empirically demonstrated the benefits of using such an approach, considerably improving the state-of-the-art in satisficing diverse planning and favorably competing with the state-of-the-art in bounded-diversity diverse planning.

For future work, in satisficing diverse planning, we intend to explore alternative ways of reformulating a planning task, aiming at tackling a specific diversity metric. For various optimal diverse planning computational problems, it is often not clear how to create a non-trivial planner for that problem at all. For example, an optD-k optimization problem,

requires to generate a set of plans that is diversity-optimal. A naive solution might require generating all possible plans first, which might be infeasible, especially in cases when the set of all plans is infinite. Focusing on such planning problems is a promising research direction.

References

2015. *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, AAAI Press.

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

Boddy, M.; Gohde, J.; Haigh, T.; and Harp, S. 2005. Course of action generation for cyber security using classical planning. In Biundo, S.; Myers, K.; and Rajan, K., eds., *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 12–21. AAAI Press.

Bryce, D. 2014. Landmark-based plan distance measures for diverse planning. In Chien et al. (2014), 56–64.

Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds. 2014. *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*. AAAI Press.

Coman, A., and Muñoz-Avila, H. 2011. Generating diverse plans using quantitative and qualitative plan distance metrics. In Burgard, W., and Roth, D., eds., *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*, 946–951. AAAI Press.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-black planning: A new systematic approach to partial delete relaxation. 221:73–114.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*, 212–221. AAAI Press.

Goldman, R. P., and Kuter, U. 2015. Measuring plan diversity: Pathologies in existing approaches and a new plan distance metric. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)* (2015), 3275–3282.

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

2018. *Ninth International Planning Competition (IPC-9): planner abstracts*.

Katz, M., and Sohrabi, S. 2019. Diversity score computation for diverse planning. `https://doi.org/10.5281/zenodo.3246578`.

Katz, M.; Lipovetzky, N.; Moshkovich, D.; and Tuisov, A. 2017. Adapting novelty to classical planning as heuristic search. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 172–180. AAAI Press.

Katz, M.; Lipovetzky, N.; Moshkovich, D.; and Tuisov, A. 2018a. Merwin planner: Mercury enchanced with novelty heuristic. In *Ninth International Planning Competition (IPC-9): planner abstracts* (2018), 53–56.

Katz, M.; Sohrabi, S.; Udrea, O.; and Winterer, D. 2018b. A novel iterative approach to top-k planning. In de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M., eds., *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*. AAAI Press.

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

Katz, M.; Sohrabi, S.; and Udrea, O. 2019a. ForbidIterative planners for top-k, top-quality, and diverse planning problems. `https://doi.org/10.5281/zenodo.3246774`.

Katz, M.; Sohrabi, S.; and Udrea, O. 2019b. Top-quality: Finding practically useful sets of best plans. In *ICAPS 2019 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*.

Katz, M. 2018. Cerberus: Red-black heuristic for planning tasks with conditional effects meets novelty heuristic and enchanced mutex detection. In *Ninth International Planning Competition (IPC-9): planner abstracts* (2018), 47–51.

Myers, K. L., and Lee, T. J. 1999. Generating qualitatively different plans through metatheoretic biases. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI 1999)*, 570–576. AAAI Press.

Nguyen, T. A.; Do, M. B.; Gerevini, A.; Serina, I.; Srivastava, B.; and Kambhampati, S. 2012. Generating diverse plans to handle unknown and partially known user preferences. 190:1–31.

Riabov, A. V.; Sohrabi, S.; Sow, D. M.; Turaga, D. S.; Udrea, O.; and Vu, L. H. 2015. Planning-based reasoning for automated large-scale data analysis. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 282–290. AAAI Press.

Riabov, A. V.; Sohrabi, S.; and Udrea, O. 2014. New algorithms for the top-k planning problem. In *ICAPS 2014 Scheduling and Planning Applications woRKshop*, 10–16.

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

Roberts, M.; Howe, A. E.; and Ray, I. 2014. Evaluating diversity in classical planning. In Chien et al. (2014), 253–261.

Shleyfman, A.; Katz, M.; Helmert, M.; Sievers, S.; and Wehrle, M. 2015. Heuristics and symmetries in classical planning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)* (2015), 3371–3377.

Sohrabi, S.; Riabov, A. V.; Udrea, O.; and Hassanzadeh, O. 2016. Finding diverse high-quality plans for hypothesis generation. In Kaminka, G. A.; Fox, M.; Bouquet, P.; Hüllermeier, E.; Dignum, V.; Dignum, F.; and van Harmelen, F., eds., *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*, 1581–1582. IOS Press.

Sohrabi, S.; Riabov, A. V.; Katz, M.; and Udrea, O. 2018. An AI planning solution to scenario generation for enterprise risk management. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, 160–167. AAAI Press.

Sohrabi, S.; Riabov, A. V.; and Udrea, O. 2016. Plan recognition as planning revisited. In Kambhampati, S., ed., *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, 3258–3264. AAAI Press.

Vadlamudi, S. G., and Kambhampati, S. 2016. A combinatorial search perspective on diverse solution generation. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI 2016)*, 776–783. AAAI Press.

Top-Quality: Finding Practically Useful Sets of Best Plans

Michael Katz and **Shirin Sohrabi** and **Octavian Udrea**
IBM T.J. Watson Research Center
1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA

Abstract

The need for finding a set of plans rather than one has been motivated by a variety of planning applications. The problem is studied in the context of both diverse and top-k planning: while diverse planning focuses on the difference between pairs of plans, the focus of top-k planning is on the quality of each individual plan. Recent work in diverse planning introduced additionally restrictions on solution quality. Naturally, there are application domains where diversity plays the major role and domains where quality is the predominant feature. In both cases, however, the amount of produced plans is somewhat an artificial constraint, and the actual number has little meaning.

Inspired by the recent work in diverse planning, we propose a new computational problem called *top-quality* planning, where solution validity is defined through plan quality bound rather than an arbitrary number of plans. Switching to bounding plan quality allows us to implicitly represent sets of plans. In particular, it makes it possible to represent sets of valid plan reorderings with one plan. We formally define the corresponding computational problem and present the first planner for that problem. We empirically demonstrate the superior performance of our approach compared to a top-k planner-based baseline, ranging from 49% increase in coverage for finding all optimal plans to 69% increase in coverage for finding all plans of quality up to 120% of optimal plan cost.

1 Introduction

While the main focus in classical planning was on producing a single plan, a variety of applications has shown the need for finding a set of plans rather than one. These applications include malware detection (Boddy et al. 2005), plan recognition as planning and its applications (Riabov et al. 2015; Sohrabi, Riabov, and Udrea 2016; Sohrabi et al. 2018; Shvo, Sohrabi, and McIlraith 2018), human team planning (Kim et al. 2018), explainable AI (Chakraborti et al. 2018), re-planning and plan monitoring (Fox et al. 2006).

The problem of finding a set of plans is studied in the context of both diverse planning (e.g., Nguyen et al. 2012) and top-k planning (e.g., Katz et al. 2018). Diverse planning focuses on the difference between pairs of plans, evaluating a set of plans by aggregating over the pairwise differences between plans in the set. Recent work in diverse planning introduced additional restrictions on solution quality, requiring each plan in the set to also be of bounded quality (Vadlamudi and Kambhampati 2016; Katz and Sohrabi 2019). Top-k planning is a generalization of cost-optimal planning. The focus of top-k planning is on the quality of each individual plan, guaranteeing that no plan of better cost exists outside the solution of a requested size.

Naturally, there are application domains where diversity plays the major role and domains where quality is the predominant feature. The latter include plan recognition (Sohrabi, Riabov, and Udrea 2016), multi-agent plan recognition (Shvo, Sohrabi, and McIlraith 2018), human team planning (Kim et al. 2018), and explainable AI (Chakraborti et al. 2018). These applications exploit top-k planners to derive a large number of plans. In these domains, though, the focus on the number of plans provided is somewhat artificial, and is intended solely to ensure that a sufficient number of plans is found. Further, ordering of actions in a plan can be of less importance in some applications. Plan recognition is one such example application. In plan recognition as planning (Sohrabi, Riabov, and Udrea 2016; Shvo, Sohrabi, and McIlraith 2018), a planning task consists of actions that explain/discard observations. There is no meaning to the order among these actions. Some specific practical applications for plan recognition are hypothesis generation (Sohrabi et al. 2016) and scenario planning advisor (Sohrabi et al. 2018). These applications use a top-k planner with a large bound on the number of required plans k, and the obtained plans are post-processed to discard reorderings and cluster similar plans. This would also apply to e.g., problems with actions that correspond to information gathering, where no particular ordering is required. The clear disadvantage of a top-k planner in such cases is that it would generate all possible orderings before proceeding to plans of a higher cost. Thus, the number of required plans used in practice is often a crude over-approximation. Further, even quite large numbers are often not sufficient to ensure that the set of plans includes enough plans of interest, since plans can easily have millions of valid reorderings. A top-k planner would have to generate all these plans before it can get to a plan of a higher cost. Diverse planners (Bryce 2014; Nguyen et al. 2012; Coman and Muñoz-Avila 2011; Roberts, Howe, and Ray 2014; Vadlamudi and Kambhampati 2016) tackle the issue by defining diversity criteria over a set of plans, but only a handful of works take the plan quality into consideration (Roberts, Howe, and Ray 2014; Vad-

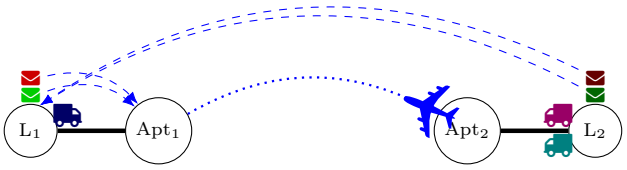


Figure 1: Example logistics task.

lamudi and Kambhampati 2016; Katz and Sohrabi 2019). While some computational problems in diverse planning do require to provide some guarantees on solution quality (Katz and Sohrabi 2019), existing diverse planners still do not provide such guarantees.

In this paper, we propose a new family of computational problems called *top-quality planning*. The objective of top-quality planning is to find and concisely represent a set of all plans of bounded quality, for a given (absolute) bound. That is, we suggest an alternative definition of solution validity, by bounding the solution quality instead of bounding the number of plans. This allows us to define an *equivalence* relation on plans and implicitly represent equivalence classes plans without knowing the exact number of plans in the set. In particular, in this work, we focus on the equivalence relation that is defined by all possible reorderings of each plan, represented by one *canonical* plan. Furthermore, we propose a first planner for *unordered* top-quality planning that iteratively finds a single plan of top quality and forbids at once all plans found so far, including all their possible reorderings. For that, we adapt a recently proposed diverse planning reformulation that forbids a single multiset of actions (Katz and Sohrabi 2019) to forbid exactly a collection of multisets. Our adaptation of the existing reformulation allows us to forbid multiple sets of plans at each iteration while preserving soundness and completeness of our approach. We empirically compare our approach to unordered top-quality planning to the only available baseline – a top-k planner with a large k bound. Our approach exhibits a superior performance, ranging from 49% increase in coverage for finding all optimal plans to 69% increase in coverage for finding all plans of cost up to 120% of optimal plan cost.

2 Preliminaries

We consider classical planning tasks in the well-known SAS$^+$ formalism (Bäckström and Nebel 1995), extended with action costs. Such *planning tasks* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ consist of $\mathcal{V}$, a finite set of finite-domain *state variables*, $\mathcal{O}$, a finite set of *actions*, s_0, an *initial state*, and $s_\star$, the *goal*. Each variable $v \in \mathcal{V}$ is associated with a finite domain $dom(v)$ of variable values. These variable, value pairs are called *facts*. A *partial assignment* p maps a subset of variables $vars(p) \subseteq \mathcal{V}$ to values in their domains. For a variable $v \in \mathcal{V}$ and partial assignment p, the value of v in p is denoted by $p[v]$ if $v \in vars(p)$ and we say $p[v]$ is *undefined* if $v \notin vars(p)$. A partial assignment s with $vars(s) = \mathcal{V}$, is called a *state*. State s is *consistent* with partial assignment p if they agree on all variables in $vars(p)$, shortly denoted by $p \subseteq s$. The product $\mathcal{S} = \prod_{v \in \mathcal{V}} dom(v)$ is called the

(load P_4 T_2 L_2)	(load P_3 T_2 L_2)	(load P_4 T_3 L_2)
(load P_3 T_2 L_2)	(load P_4 T_2 L_2)	(load P_3 T_3 L_2)
(drive T_2 L_2 Apt_2)	(drive T_2 L_2 Apt_2)	(drive T_3 L_2 Apt_2)
(unload P_4 T_2 Apt_2)	(unload P_4 T_2 Apt_2)	(unload P_4 T_3 Apt_2)
(unload P_3 T_2 Apt_2)	(unload P_3 T_2 Apt_2)	(unload P_3 T_3 Apt_2)
(load P_2 T_1 L_1)	(load P_2 T_1 L_1)	(load P_2 T_1 L_1)
(load P_1 T_1 L_1)	(load P_1 T_1 L_1)	(load P_1 T_1 L_1)
(load P_3 A Apt_2)	(load P_3 A Apt_2)	(load P_3 A Apt_2)
(load P_4 A Apt_2)	(load P_4 A Apt_2)	(load P_4 A Apt_2)
(fly A Apt_2 Apt_1)	(fly A Apt_2 Apt_1)	(fly A Apt_2 Apt_1)
(unload P_3 A Apt_1)	(unload P_3 A Apt_1)	(unload P_3 A Apt_1)
(unload P_4 A Apt_1)	(unload P_4 A Apt_1)	(unload P_4 A Apt_1)
(drive T_1 L_1 Apt_1)	(drive T_1 L_1 Apt_1)	(drive T_1 L_1 Apt_1)
(load P_4 T_1 Apt_1)	(load P_4 T_1 Apt_1)	(load P_4 T_1 Apt_1)
(load P_3 T_1 Apt_1)	(load P_3 T_1 Apt_1)	(load P_3 T_1 Apt_1)
(unload P_2 T_1 Apt_1)	(unload P_2 T_1 Apt_1)	(unload P_2 T_1 Apt_1)
(unload P_1 T_1 Apt_1)	(unload P_1 T_1 Apt_1)	(unload P_1 T_1 Apt_1)
(drive T_1 Apt_1 L_1)	(drive T_1 Apt_1 L_1)	(drive T_1 Apt_1 L_1)
(unload P_4 T_1 L_1)	(unload P_4 T_1 L_1)	(unload P_4 T_1 L_1)
(unload P_3 T_1 L_1)	(unload P_3 T_1 L_1)	(unload P_3 T_1 L_1)
π_a	π_b	π_c

Figure 2: Three example cost-optimal plans for the example task.

state space of planning task Π. s_0 is a state and $s_\star$ is a partial assignment. A state s is called a *goal state* if $s_\star \subseteq s$ and the set of all goal states is denoted by $\mathcal{S}_{s_\star}$. Each action o in $\mathcal{O}$ is a pair $\langle pre(o), \mathit{eff}(o) \rangle$ where $pre(o)$ is a partial assignment called *precondition* and $\mathit{eff}(o)$ is a partial assignment called *effect*. Further, o has an associated natural number $cost(o)$, called *cost*. An action o is applicable in state s if $pre(o) \subseteq s$. Applying action o in state s results in a state denoted by $s[\![o]\!]$ where $s[\![o]\!][v] = \mathit{eff}(o)[v]$ for all $v \in vars(\mathit{eff})$ and $= s[\![o]\!][v] = s[v]$ for all other variables. An action sequence $\pi = \langle o_1, \cdots, o_n \rangle$ is applicable in state s if there are states $s_0, \cdots, s_n$ such that o_i is applicable in s_{i-1} and $s_{i-1}[\![o_i]\!] = s_i$ for $0 \leq i \leq n$. We denote s_n by $s[\![\pi]\!]$. For convenience we often write $o_1, \cdots, o_n$ instead of $\langle o_1, \cdots, o_n \rangle$. An action sequence with $s_0[\![\pi]\!] \in \mathcal{S}_{s_\star}$ is called a *plan*. The cost of a plan π, denoted by $cost(\pi)$ is the summed cost of the actions in the plan. For a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, the set of all plans is denoted by $\mathcal{P}_\Pi$. A plan π is *optimal* if its cost is minimal among all plans in $\mathcal{P}_\Pi$. For a plan π, we denote by $\mathrm{MS}(\pi)$ the multiset[1] of actions in π. Note that two different plans π and π' can have $\mathrm{MS}(\pi) = \mathrm{MS}(\pi')$. We call such plans *reordering* of each other. Reoderings of actions of a plan that are plans are called *valid* reorderings.

In this paper, we will use a logistics task, depicted in Figure 1, as our running example. This task has two cities, with two locations each, L_1 and L_2, three trucks, T_1 (left), and T_2, T_3 (right), that can drive within their cities, one airplane, A, that can fly between the airport locations, Apt_1 and Apt_2, and four packages, P_1 to P_4, that need to be transported from their initial locations to some specified goal locations. The initial and goal locations of all objects are shown in Figure 1 and marked with dashed arrows. Assuming all actions hav-

[1] A set with possible multiple occurences of the same element.

ing unit cost, a cost-optimal plan for this task will consists of 20 actions. Example plans π_a, π_b, and π_c are depicted in Figure 2.

Given a plan π, it is sometimes possible to obtain a different plan of equivalent cost without solving the planning task again. Two of such possible ways: action reordering and deriving symmetric plans, are exploited by state-of-the-art top-k planners (Katz et al. 2018). While action reordering is performed using search and may be time consuming, symmetric plans can be obtained using structural symmetries (Shleyfman et al. 2015). Structural symmetries are permutations of variable values and actions that induce automorphisms of the state transition graph. Here, we present the definition of structural symmetries for SAS$^+$ as was given by Sievers et al. (2017).

Definition 1 (structural symmetry) *For a* SAS$^+$ *planning task* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$*, let* F *be the set of* Π*'s facts, i. e. pairs* $\langle v, d \rangle$ *with* $v \in \mathcal{V}$, $d \in dom(v)$*. A* structural symmetry *for* Π *is a permutation* $\sigma : \mathcal{V} \cup F \cup \mathcal{O} \to \mathcal{V} \cup F \cup \mathcal{O}$*, where:*

1. $\sigma(\mathcal{V}) = \mathcal{V}$ *and* $\sigma(F) = F$ *such that* $\sigma(\langle v, d \rangle) = \langle v', d' \rangle$ *implies* $v' = \sigma(v)$*;*
2. $\sigma(\mathcal{O}) = \mathcal{O}$ *such that for* $o \in \mathcal{O}$*,* $\sigma(pre(o)) = pre(\sigma(o))$*,* $\sigma(\mathit{eff}(o)) = \mathit{eff}(\sigma(o))$*,* $cost(\sigma(o)) = cost(o)$*;*
3. $\sigma(s_\star) = s_\star$*;*

where $\sigma(\{x_1, \ldots, x_n\}) := \{\sigma(x_1), \ldots, \sigma(x_n)\}$*, and* $s' := \sigma(s)$ *is the partial state obtained from the partial state* s *s.t. for all* $v \in vars(s)$*,* $\sigma(\langle v, s[v] \rangle) = \langle v', d' \rangle$ *implies* $s'[v'] = d'$*.*

A structural symmetry σ *stabilizes* the state s if $\sigma(s) = s$. Given a plan $\pi = o_1 \ldots o_n$ and a structural symmetry σ that stabilizes the initial state, applying the permutation σ to each action in the plan results in a necessarily valid plan $\sigma(\pi) = \sigma(o_1) \ldots \sigma(o_n)$ of the same cost. Note that $\sigma(\pi)$ is not a reordering of π, since σ may map actions from π to actions outside of π.

In our example, the structural symmetries can detect symmetries between two of the trucks T_2 and T_3, between the two packages that are initially in L_1, and between the two packages that are initially in L_2. Thus, structural symmetries can be used to obtain additional plans from a given plan. In our example, the plan π_c in Figure 2 can be obtained from π_a using the symmetry between the trucks T_2 and T_3. Note that these two plans use different actions and thus are not reorderings of each other. The plan π_b, on the other hand, is a reordering of π_a, changing the order between the first two actions. These two plans are not symmetric, since mapping the action (load P_4 T_2 L_2) to (load P_3 T_2 L_2) would also require mapping (unload P_4 T_2 L_2) to (unload P_3 T_2 L_2). Naturally, there exist pairs of plans that are both symmetric and reordering of each other. There are 6602112 cost-optimal plans in our example, half of them are reorderings of the plan π_a and the other half are reordering of π_c.

Lastly, the *top-k planning problem* (Sohrabi et al. 2016; Katz et al. 2018) is defined as follows.

Definition 2 (top-k planning problem) *Given a planning task* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ *and a natural number* k*, find a set of plans* $P \subseteq \mathcal{P}_\Pi$ *such that:*

(i) for all plans $\pi \in P$*, if there exists a plan* π' *for* Π *with* $cost(\pi') < cost(\pi)$*, then* $\pi' \in P$*, and*

(ii) $|P| \leq k$*, with* $|P| < k$ *implying* $P = \mathcal{P}_\Pi$*.*

An instance of the top-k planning problem $\langle \Pi, k \rangle$*, is called solvable if* $|P| = k$ *and unsolvable if* $|P| < k$*.*

The objective of top-k planning is finding k plans of lowest costs for a planning task Π and thus optimal planning is the special case of top-1 planning.

3 Top-quality Planning

We start by formally defining the top-quality planning problem as the problem of finding all plans of bounded quality.

Definition 3 (top-quality planning problem) *Given a planning task* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ *and a natural number* q*, find the set of plans* $P = \{\pi \in \mathcal{P}_\Pi \mid cost(\pi) \leq q\}$*.*

The top-quality planning problem is well-defined and always has a solution. Note that one can exploit existing tools for top-k planning to derive solutions to the top-quality planning problem, by setting k to a large value and adding another stopping criteria, once a plan π of $cost(\pi) > q$ was obtained. In such cases, P would explicitly contain all plans with $cost(\pi) \leq q$. This was done by Vadlamudi and Kambhampati (2016) as the first step in their algorithm, although they do not formally define the top-quality problem. These explicit sets of plans can get prohibitively large. Further, some of the plans in that set, although different as sequences of actions, could be considered equivalent from the underlying application perspective. If, in addition, it would be possible to escape the need for generating all these equivalent plans, the performance of the planners could improve significantly.

Let N be some equivalence relation on the set of plans $\mathcal{P}_\Pi$. For a plan $\pi \in \mathcal{P}_\Pi$, by $N[\pi]$ we denote the equivalence class of π, which is a set of all plans that are equivalent to π under N. Slightly abusing the notation, for a set of plans P, by $N[P]$ we denote the union of the equivalence classes $\bigcap_{\pi \in P} N[\pi]$. Using that equivalence relation, we can define the quotient top-quality problem as follows.

Definition 4 (quotient top-quality planning problem) *Given a planning task* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$*, an equivalence relation* N *over its set of plans* $\mathcal{P}_\Pi$*, and a natural number* q*, find a set of plans* $P \subseteq \mathcal{P}_\Pi$ *such that* $\bigcup_{\pi \in P} N[\pi]$ *is the solution to the top-quality planning problem.*

For equivalence relations that preserve plan cost the quotient top-quality planning problem always has a solution. Note that solutions to top-quality planning are solutions to the quotient top-quality planning under the identity equivalence relation. Further, while there is one possible solution to the top-quality planning problem, there can be many solutions to a quotient top-quality problem, defined by representatives of each equivalence class. Further, nothing in our definition prevents a solution from including more than one plan per equivalence class, the only restriction is that all equivalence classes have to be represented.

In this paper, we focus on one specific equivalence relation, considering two plans to be equivalent if their action multi-sets are. Formally, we consider the equivalence relation

$$\mathrm{U}_\Pi = \{(\pi, \pi') \mid \pi, \pi' \in \mathcal{P}_\Pi, \mathrm{MS}(\pi) = \mathrm{MS}(\pi')\}.$$

Thus, the main computational problem we consider in this paper is as follows.

Definition 5 (unordered top-quality planning problem) *Given a planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ and a natural number q, find a set of plans $P \subseteq \mathcal{P}_\Pi$ such that P is a solution to the quotient top-quality planning problem under the equivalence relation U_Π.*

Note that while the solution to the top-quality planning problem can be obtained from a solution to the unordered top-quality planning problem, using a simple algorithm that generates all possible valid reordering for each plan in the solution, this is not the focus of current work. Focusing on the unordered top-quality planning problem allows us to generate reorderings of the same plan only if and when these reorderings are actually needed.

4 Computation of Top-quality Plans

In order to obtain a solver for the computational problem specified above, we take an approach similar to Katz et al. (2018), and iteratively generate plans using an existing cost-optimal planner, and construct planning tasks with a reduced set of plans, by forbidding exactly the plans found so far. In contrast to Katz et al. (2018), we forbid not only a specific plan, but also all its possible reorderings. In order to achieve that, we thus instead of forbidding plans as sequences of actions, forbid plans as multi-sets. To be able to do that, we need to come up with a reformulation of a planning task that forbids all plans with the exact number of appearances for each action. Similar reformulation was recently suggested by Katz and Sohrabi (2019) for diverse planning. The reformulation can forbid a single multi-set, and thus for a set of plans, the union of their multi-sets was forbidden in each consecutive iteration. That way, possibly additional plans were forbidden. For diverse planning, that did not pose a problem. In our case, however, we need to ensure that we forbid *exactly* the set of plans that were previously found. For that, in what follows, we adapt the reformulation of Katz and Sohrabi (2019) accordingly.

Alternatively, the reformulation of Katz and Sohrabi (2019) can be used directly, creating a sequence of planning tasks, similarly to the way it was done in top-k planning (Katz et al. 2018). This. however, poses two problems: the reformulated planning task size grows fast with each iteration, and, as in the iterative top-k planner, the mapping between the reformulated and original actions must be constantly maintained.

In this work, at each iteration we reformulate the original planning task to forbid all plans found so far. In this case, we do not need to maintain the mapping between the reformulated and original actions and keep the reformulated task size smaller. In the rest of this section we adapt the definition of Katz and Sohrabi (2019) to a set of plans (as multi-sets), present an algorithm that exploits the adapted definition to derive top-quality solutions, and prove its soundness and completeness. We start by presenting the definition.

4.1 Forbidding a Plan as a Multi-set of Actions

Slightly simplifying the definition of Katz and Sohrabi (2019), we present a task reformulation that ignores orders between actions in a plan and thus also forbids all possible reorderings of a given plan, as well as all sub-plans.

Definition 6 *Let $\langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task and π be a plan. The task $\Pi^-_\pi = \langle \mathcal{V}', \mathcal{O}', s'_0, s'_\star \rangle$ is defined as follows.*

- *$\mathcal{V}' = \mathcal{V} \cup \{\overline{v}\} \cup \{\overline{v}_o \mid o \in \pi\}$, with $\overline{v}$ being a binary variable, and $dom(\overline{v}_o) = \{0, \dots, m_o\}$, where m_o is the number of occurences of o in π,*
- *$\mathcal{O}' = \{o^e \mid o \in \mathcal{O} \setminus \pi\} \cup \bigcup_{i=0}^{m_o} \{o^f_i \mid o \in \pi\}$, where*
  $pre(o^e) = pre(o), \quad \mathit{eff}(o^e) = \mathit{eff}(o) \cup \{\langle \overline{v}, 0 \rangle\}$,
  $pre(o^f_i) = pre(o) \cup \{\langle \overline{v}_o, i \rangle\}$,
  for $0 \le i < m_o$, $\mathit{eff}(o^f_i) = \mathit{eff}(o) \cup \{\langle \overline{v}_o, i{+}1 \rangle\}$,
  $\mathit{eff}(o^f_{m_o}) = \mathit{eff}(o) \cup \{\langle \overline{v}, 0 \rangle\}$, and
  $cost'(o^e) = cost'(o^f_i) = cost(o)$, $0 \le i \le m_o$,
- *$s'_0[v] = s_0[v]$ for all $v \in \mathcal{V}$, $s'_0[\overline{v}] = 1$, and $s'_0[\overline{v}_o] = 0$ for all $o \in \pi$, and*
- *$s'_\star[v] = s_\star[v]$ for all $v \in \mathcal{V}$ s.t. $s_\star[v]$ defined, and $s'_\star[\overline{v}] = 0$.*

The semantics of the reformulation is as follows. The variable $\overline{v}$ starts from the value 1 and switches to 0 when an action is applied that is not from plan π treated as a multi-set. Once a value 0 is reached indicating a deviation from plan π, it cannot be switched back to 1. The finite-domain variables $\overline{v}_o$ encode the number of applications of the action o. The actions o^f_i are copies of the action o in π, counting the number of applications of o, as long as the number is not higher than the number of times it appears in π. Once the number of applications exceeds m_o, $\overline{v}$ is set to 0.

4.2 Forbidding Multiple Plans Exactly

In order to forbid multiple plans, the greedy approach of Katz and Sohrabi (2019) forbids the super-set of these plans by taking a super-set of the multi-sets representing the plans. In our case, when optimality is required, we cannot follow the same approach. Instead, we present a reformulation that forbids exactly these plans and their sub-plans, and the possible reorderings. Our reformulation extends the one in Definition 6, by introducing a binary variable for each plan, encoding whether the plan was deviated from.

Definition 7 *Let $\langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ be a planning task, P be a set of plans, and $\mathcal{O}_P = \{o \mid o \in \pi, \pi \in P\}$. The task $\Pi^-_P = \langle \mathcal{V}', \mathcal{O}', s'_0, s'_\star \rangle$ is defined as follows.*

- *$\mathcal{V}' = \mathcal{V} \cup \{\overline{v}_\pi \mid \pi \in P\} \cup \{\overline{v}_o \mid o \in \mathcal{O}_P\}$, with $\overline{v}_\pi$ being binary variables, and $dom(\overline{v}_o) = \{0, \dots, m_o\}$, where $m_o = \max_{\pi \in P}\{m^\pi_o\}$ and m^π_o is the number of occurrences of o in π,*

- $\mathcal{O}' = \{o^e \mid o \in \mathcal{O} \setminus \mathcal{O}_P\} \cup \{o_i^f \mid o \in \mathcal{O}_P, 0 \leq i \leq m_o\}$, *where*
  $pre(o^e) = pre(o)$, $\mathit{eff}(o^e) = \mathit{eff}(o) \cup \{\langle \overline{v}_\pi, 0\rangle \mid \pi \in P\}$,
  $pre(o_i^f) = pre(o) \cup \{\langle \overline{v}_o, i\rangle\}$,
  $\mathit{eff}(o_i^f) = \mathit{eff}(o) \cup \{\langle \overline{v}_o, i{+}1\rangle\} \cup \{\langle \overline{v}_\pi, 0\rangle \mid i = m_{op}^\pi\}$ *for* $0 \leq i < m_o$,
  $\mathit{eff}(o_{m_o}^f) = \mathit{eff}(o) \cup \{\langle \overline{v}_\pi, 0\rangle \mid \pi \in P\}$, *and*
  $cost'(o^e) = cost'(o_i^f) = cost(o)$, $0 \leq i \leq m_o$,
- $s_0'[v] = s_0[v]$ *for all* $v \in \mathcal{V}$, $s_0'[\overline{v}_\pi] = 1$ *for all* $\pi \in P$, *and* $s_0'[\overline{v}_o] = 0$ *for all* $o \in \mathcal{O}_P$, *and*
- $s_\star'[v] = s_\star[v]$ *for all* $v \in \mathcal{V}$ *s.t.* $s_\star[v]$ *defined, and* $s_\star'[\overline{v}_\pi] = 0$ *for all* $\pi \in P$.

4.3 Using the Reformulation

Algorithm 1 exploits the reformulation in Definition 7 to find a solution to the unordered top-quality planning problem. The algorithm incrementally finds the set P of top quality plans. Starting with the empty set $P = \emptyset$ and assuming $\Pi_\emptyset^- = \Pi$, we use an optimal planner iteratively to find an optimal plan π to the planning task Π_P^-. Once a plan is found, it is added to the set of found plans P. Then, the new reformulation Π_P^- is constructed from Π for the next iteration. The algorithm stops when a plan π is generated such that $cost(\pi) > q$. Note that the algorithm results in a set P of sequential plans, with no two plans being reorderings of each other. Similarly to Katz et al. (2018), at each iteration, after the plan π was found, we use structural symmetries to generate from π additional plans that are symmetric (Shleyfman et al. 2015) to π, and add these that are not reorderings of π to the set P. Finally, since the first step results in an optimal plan, the quality can be defined relatively to the cost of the optimal plan rather than an absolute number.

Theorem 1 *Algorithm 1 is sound and complete for unordered top-quality planning when using cost-optimal planners that find shortest (in the number of actions) cost-optimal plans.*

Proof: Let P be the set of plans returned by Algorithm 1 and let π_f be the plan found when the algorithm breaks. Since π_f is an optimal plan to Π_P^- and $cost(\pi_f) > q$, we need to show that Π_P^- forbids exactly the plans in $\mathrm{U}_\Pi[P]$. For a plan $\pi \in P$, Π_P^- has a variable $\overline{v}_\pi$ that reaches its goal value only when the number of applications of some action exceeds the number of appearances of that action in π. Thus, π is not a plan for Π_P^-. Since Definition 7 treats plans as multi-sets, this is true also for all $\pi' \in \mathrm{U}_\Pi[\pi]$.

Let $P_1, \ldots, P_n$ denote the sets of plans at the beginning of each algorithm iteration and let $\pi_1, \ldots, \pi_n = \pi_f$ be the optimal plans found by the algorithm in these iteration, with π_i being an optimal plan to $\Pi_{P_i}^-$. Let π be a plan for Π such that $cost(\pi) \leq q$. If $\pi \notin \mathrm{U}_\Pi[P]$, there exists k such that π is a plan for $\Pi_{P_k}^-$, but not for $\Pi_{P_{k+1}}^-$. Let $P' = P_{k+1} \setminus P_k$ be the plans forbidden in $\Pi_{P_{k+1}}^-$ but not in $\Pi_{P_k}^-$. Then, there exists $\pi' \in P'$ such that $\mathrm{MS}(\pi) \subseteq \mathrm{MS}(\pi')$. If $\mathrm{MS}(\pi) = \mathrm{MS}(\pi')$, then $\pi \in P$ and we are done. Assume that $\mathrm{MS}(\pi)$ is a proper subset of $\mathrm{MS}(\pi')$. Note that π' is a reordering of a plan that is symmetric to π_k, which was the optimal plan found for $\Pi_{P_k}^-$. Assuming that our optimal planner finds shorter optimal plans before longer ones, a plan π for $\Pi_{P_k}^-$ would be found before π_k, contradicting the assumption that $\mathrm{MS}(\pi)$ is a proper subset of $\mathrm{MS}(\pi')$. □

Algorithm 1 Iterative unordered top-quality planning.

Input: Planning task Π, quality bound q
  $P \leftarrow \emptyset$
  $\Pi' \leftarrow \Pi$
  while True **do**
    $\pi \leftarrow$ optimal plan to Π'
    if $cost(\pi) > q$ **then**
      break
    end if
    $P \leftarrow P \cup \{\pi\} \cup \{\pi' \mid \pi'$ is symmetric to $\pi, \pi' \notin \mathrm{U}_\Pi[\pi]\}$
    $\Pi' \leftarrow \Pi_P^-$ according to Definition 7
  end while
  return P

5 Experimental Evaluation

In order to evaluate the feasibility of our suggested approach for unordered top-quality planning, we have implemented our approach as part of the ForbidIterative planners collection (Katz, Sohrabi, and Udrea 2019), which is implemented on top of the Fast Downward planning system (Helmert 2006). The collection, among other, includes the implementation of the iterative top-k planner (Katz et al. 2018). The experiments were performed on Intel(R) Xeon(R) CPU E7-8837 @2.67GHz machines, with the time and memory limit of 30min and 2GB, respectively. The benchmark set consists of all STRIPS benchmarks from optimal tracks of International Planning Competitions (IPC) 1998-2018, a total of 1797 tasks in 64 domains. Our baseline for the comparison is a simple approach, using a top-k planner with a large k value, 10^9, stopping if a plan of quality above the bound was reached. We use *NaiveS*, the best perfoming configuration of the iterative top-k planner (Katz et al. 2018), that exploits both symmetries and plan reorderings. The purpose of setting k to a large number is to allow the top-k planner to exploit the entire 30min time interval. Among tasks solved, the largest number of plans found by the top-k planner was 60480. For tasks not solved, the maximal number of plans found by the top-k planner was 767501. Note that reading and writing such large amounts of plans is time consuming by itself. For each task, the quality bound is computed using the cost of the first found (optimal) plan, multiplied by a constant[2]. We experiment with four different quality bound multipliers, namely $q_m = 1.0$ (optimal plans only), 1.05, 1.1, and 1.2 of the optimal plan cost. For larger quality bounds, both approaches had low coverage, and thus we do not report these results. Note, q can be any natural number as mentioned in Definition 5.

[2]This is not an overhead, as at least one optimal planner run needs to be performed anyway.

	$q_m = 1.00$		$q_m = 1.05$		$q_m = 1.10$		$q_m = 1.20$	
Coverage	K-tq	tq	K-tq	tq	K-tq	tq	K-tq	tq
airport	7	**21**	7	**18**	6	**17**	6	**17**
blocks	16	**17**	16	**17**	10	**13**	8	**9**
data-network18	0	**1**	0	0	0	0	0	0
depot	2	2	2	2	0	**2**	0	**1**
driverlog	5	**9**	5	**9**	1	**7**	1	**4**
floortile11	0	**2**	0	**2**	0	0	0	0
ged14	5	**7**	5	**7**	5	**7**	5	**7**
gripper	1	**4**	1	**3**	0	**2**	0	**2**
logistics00	3	**16**	3	**13**	1	**10**	0	**6**
logistics98	0	**4**	0	**2**	0	**1**	0	0
miconic	18	**27**	18	**26**	11	**16**	10	**12**
movie	0	**1**	0	**1**	0	**1**	0	0
mprime	18	**19**	18	**19**	18	**19**	6	**11**
mystery	20	20	20	20	20	20	13	**15**
nomystery11	9	**13**	7	**11**	4	**8**	2	**5**
openstacks08	0	**2**	0	**2**	0	**2**	0	**2**
parcprinter08	6	**15**	5	**12**	5	**12**	5	**11**
parcprinter11	3	**11**	2	**8**	2	**8**	2	**7**
pegsol08	21	**23**	21	**23**	21	**22**	8	**17**
pegsol11	8	**13**	8	**13**	8	**12**	2	**5**
pipes-notank	5	**11**	5	**11**	3	**7**	1	**4**
pipes-tank	2	**4**	2	**4**	2	**4**	1	1
psr-small	37	**46**	26	**40**	22	**36**	16	**24**
rovers	3	**6**	3	**6**	2	**4**	0	**3**
satellite	2	**5**	2	**5**	1	1	0	**1**
scanalyzer08	4	**5**	4	**5**	4	4	3	3
scanalyzer11	1	**2**	1	**2**	1	1	1	1
spider18	5	5	3	**4**	0	0	0	0
storage	8	**14**	8	**14**	7	**11**	6	**7**
tetris14	1	**2**	1	**2**	1	**2**	0	**1**
tidybot11	5	**7**	2	**5**	1	**3**	1	1
tpp	4	**6**	4	**5**	2	**5**	2	**5**
transport08	6	**7**	1	1	1	1	0	0
transport14	0	**1**	0	0	0	0	0	0
trucks	1	**2**	1	**2**	0	**1**	0	0
visitall11	8	8	7	7	5	**6**	5	5
woodwork08	3	**8**	2	**6**	1	**4**	0	**2**
woodwork11	0	**3**	0	**1**	0	0	0	0
zenotravel	7	7	7	7	4	**5**	3	**4**
Sum other	25	25	23	23	21	21	18	18
Sum (1797)	269	**401**	240	**358**	190	**295**	125	**211**

Table 1: The coverage results comparing to top quality planning via top-k planning, for various quality bounds.

Table 1 depicts the per-domain summed coverage, comparing our technique, tq, to the baseline, K-tq, for four quality bound multipliers. Each task gets a coverage of 1 if and only if the planner proved there is no other plan within quality bound, by either finding a plan above the bound or proving there are no other plans. Note first that out of the 64 domains, there are 18 domains where all optimal plans could not be found for any tasks, with any approach. There are 7 more domains where there is no difference in coverage between the baseline and our approach, for all tested quality bounds. These 25 domains are summarized in the *Sum other* row of Table 1. Out of the remaining 39 domains, the coverage never gets worse and it gets better (often significantly better) for at least one of the tested quality bounds. Extreme examples are AIRPORT, LOGISTICS00, and PSR-SMALL where the increase in coverage for some quality

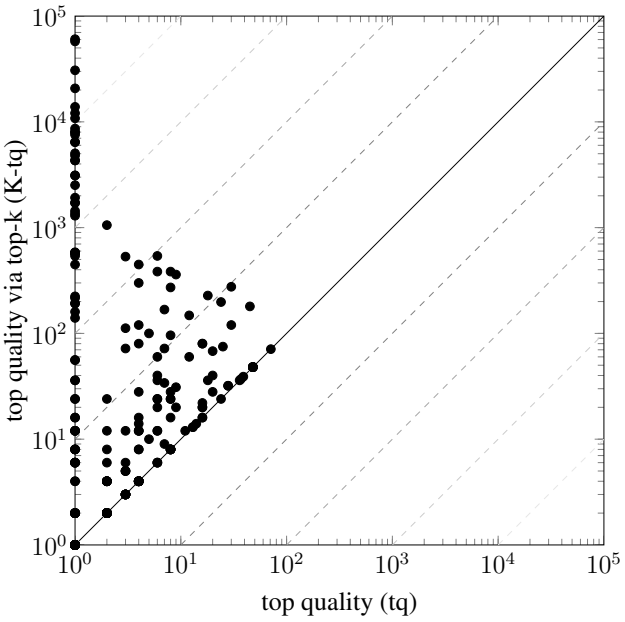


Figure 3: Per-task comparison of the solution encoding size.

bounds is by 10 instances or more. Overall, there is a clear benefit of the suggested approach over the baseline.

Another benefit of our approach is a compact representation of the solution. Figure 3 shows a per-task comparison of the number of plans in the solution for each of the approaches, for the quality bound multiplier $q_m = 1.0$, for tasks solved by both approaches. First, out of the total of 263 such tasks, there are 111 tasks on the diagonal. Out of the remaining 152 tasks (all above the diagonal), 73 tasks have a single optimal plan found by our approach, while the baseline needs to find multiple optimal plans, which are all reorderings of the same plan, with the maximal number of 60480 reorderings found. When the number of valid reordering is larger, the baseline approach fails before being able to find all optimal plans.

Finally, Figure 4 compares the reformulated task size of our approach to the baseline one. We compare the last generated task reformulation, for tasks solved by both approaches, for the quality bound multiplier $q_m = 1.0$. The task size is measured here by the number of facts, i. e., variable value pairs. While the larger tasks are not necessarily harder for a classical planner, this is usually the case. Our experiments clearly show that our approach creates tasks of sizes almost two orders of magnitude smaller than the baseline approach.

6 Conclusions and Future Work

In this work we have shown a way of obtaining all plans of bounded solution quality, representing plan reorderings implicitly and thus escaping the need for counting plans. We have presented a novel reformulation of a planning task that forbids exactly the set of given plans, their reorderings, and all subplans thereof. We have formally defined the family of computational problems in top-quality planning and

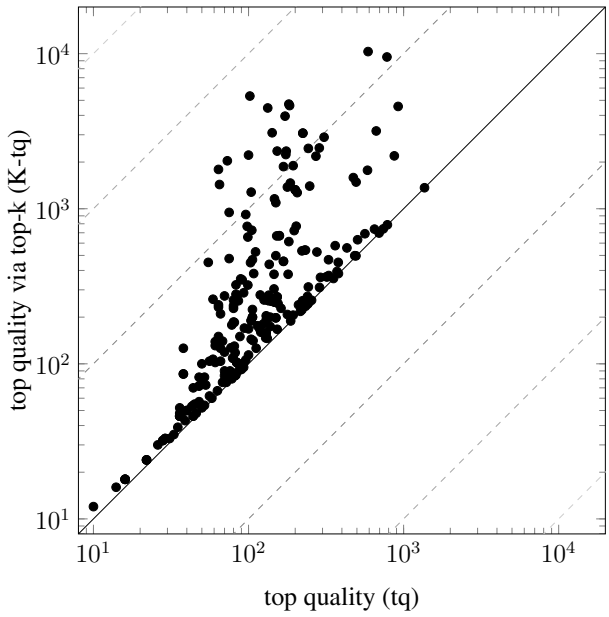


Figure 4: Final constructed task size in terms of the number of facts.

have implemented a first planner for unordered top-quality planning. The planner, exploiting the new reformulation, has empirically shown to perform significantly better than the straightforward approach of exploiting top-k planners with a large bound k, as it is often done in practice.

For future work, one promising direction is exploring the use of top-quality instead of top-k planners in planning applications. Another possible direction is creating a top-k planner based on the unordered top-quality planner, exploiting the more compact task representation. Further, (unordered) top-quality planners can be used to obtain solutions to diverse planning, when solution cost is also considered (Vadlamudi and Kambhampati 2016).

References

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

Boddy, M.; Gohde, J.; Haigh, T.; and Harp, S. 2005. Course of action generation for cyber security using classical planning. In Biundo, S.; Myers, K.; and Rajan, K., eds., *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 12–21. AAAI Press.

Bryce, D. 2014. Landmark-based plan distance measures for diverse planning. In Chien et al. (2014), 56–64.

Chakraborti, T.; Fadnis, K. P.; Talamadupula, K.; Dholakia, M.; Srivastava, B.; Kephart, J. O.; and Bellamy, R. K. E. 2018. Visualizations for an explainable planning agent. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)* (2018), 5820–5822.

Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds. 2014. *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*. AAAI Press.

Coman, A., and Muñoz-Avila, H. 2011. Generating diverse plans using quantitative and qualitative plan distance metrics. In Burgard, W., and Roth, D., eds., *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*, 946–951. AAAI Press.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*, 212–221. AAAI Press.

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

2018. *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)*, IJCAI.

Katz, M., and Sohrabi, S. 2019. Reshaping diverse planning: Let there be light! In *ICAPS 2019 Workshop on Heuristics and Search for Domain-independent Planning (HSDIP)*.

Katz, M.; Sohrabi, S.; Udrea, O.; and Winterer, D. 2018. A novel iterative approach to top-k planning. In de Weerdt, M.; Koenig, S.; Röger, G.; and Spaan, M., eds., *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*. AAAI Press.

Katz, M.; Sohrabi, S.; and Udrea, O. 2019. ForbidIterative planners for top-k, top-quality, and diverse planning problems. `https://doi.org/10.5281/zenodo.3246774`.

Kim, J.; Woicik, M. E.; Gombolay, M. C.; Son, S.; and Shah, J. A. 2018. Learning to infer final plans in human team planning. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI 2018)* (2018), 4771–4779.

Nguyen, T. A.; Do, M. B.; Gerevini, A.; Serina, I.; Srivastava, B.; and Kambhampati, S. 2012. Generating diverse plans to handle unknown and partially known user preferences. 190:1–31.

Riabov, A. V.; Sohrabi, S.; Sow, D. M.; Turaga, D. S.; Udrea, O.; and Vu, L. H. 2015. Planning-based reasoning for automated large-scale data analysis. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 282–290. AAAI Press.

Roberts, M.; Howe, A. E.; and Ray, I. 2014. Evaluating diversity in classical planning. In Chien et al. (2014), 253–261.

Shleyfman, A.; Katz, M.; Helmert, M.; Sievers, S.; and Wehrle, M. 2015. Heuristics and symmetries in classical planning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, 3371–3377. AAAI Press.

Shvo, M.; Sohrabi, S.; and McIlraith, S. A. 2018. An AI planning-based approach to the multi-agent plan recognition problem. In Bagheri, E., and Cheung, J. C., eds., *Proceedings of the 31st Canadian Conference on Artificial In-*

telligence (CAI 2018), volume 10832, 253–258. Springer-Verlag.

Sievers, S.; Wehrle, M.; Helmert, M.; and Katz, M. 2017. Strengthening canonical pattern databases with structural symmetries. In Fukunaga, A., and Kishimoto, A., eds., *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS 2017)*, 91–99. AAAI Press.

Sohrabi, S.; Riabov, A. V.; Udrea, O.; and Hassanzadeh, O. 2016. Finding diverse high-quality plans for hypothesis generation. In Kaminka, G. A.; Fox, M.; Bouquet, P.; Hüllermeier, E.; Dignum, V.; Dignum, F.; and van Harmelen, F., eds., *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*, 1581–1582. IOS Press.

Sohrabi, S.; Riabov, A. V.; Katz, M.; and Udrea, O. 2018. An AI planning solution to scenario generation for enterprise risk management. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence (AAAI 2018)*, 160–167. AAAI Press.

Sohrabi, S.; Riabov, A. V.; and Udrea, O. 2016. Plan recognition as planning revisited. In Kambhampati, S., ed., *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI 2016)*, 3258–3264. AAAI Press.

Vadlamudi, S. G., and Kambhampati, S. 2016. A combinatorial search perspective on diverse solution generation. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI 2016)*, 776–783. AAAI Press.

Merge-and-Shrink Task Reformulation for Classical Planning

Álvaro Torralba
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany
torralba@cs.uni-saarland.de

Silvan Sievers
Basel University
Switzerland
silvan.sievers@unibas.ch

Abstract

The performance of domain-independent planning systems heavily depends on how the planning task has been modeled. This makes task reformulation an important tool to get rid of unnecessary complexity and increase the robustness of planners with respect to the model chosen by the user. In this paper, we represent tasks as factored transition systems (FTS), and use the merge-and-shrink (M&S) framework for task reformulation for optimal and satisficing planning. We prove that the flexibility of the underlying representation makes the M&S reformulation methods more powerful than the counterparts based on the more popular finite-domain representation. We adapt delete-relaxation and M&S heuristics to work on the FTS representation and evaluate the impact of our reformulation.

Introduction

Classical planning deals with the problem of finding a sequence of actions that achieve a set of goals, given a model of the world that describes an initial state and a set of available actions. For representing the problem, different planning formalisms can be used, the most common being STRIPS or finite-domain representation (FDR). The choice of formalism does not change the complexity of the problem, which is PSPACE-complete (Bylander 1994; Bäckström and Nebel 1995). However, it may impact the so-called accidental complexity, when the structure of the task is disguised by how it is encoded (Haslum 2007). Accidental complexity can be dealt with by reformulating the planning task prior to solving it. There are several reformulation methods based on, e.g., downward-refinable abstractions (Haslum 2007) or tunnel macros (Coles and Coles 2010), which can be combined to reduce the size of FDR tasks (Tozicka et al. 2016).

Merge-and-Shrink (M&S) is a general framework to generate abstractions, originally defined in the model-checking area (Dräger, Finkbeiner, and Podelski 2006; 2009), that can be used to derive an admissible heuristic (Helmert, Haslum, and Hoffmann 2007; Helmert et al. 2014) and/or detect unsolvability (Hoffmann, Kissmann, and Torralba 2014). Further work on the topic noticed that this can be understood as applying transformations to a set of transition systems (Sievers, Wehrle, and Helmert 2014) and hence as a method to transform planning tasks in the factored transition system (FTS) representation (Torralba and Kissmann 2015). However, these methods perform the search on an FDR task, only using M&S to derive heuristics or remove irrelevant actions.

In this paper, we use M&S as a task reformulation method on FTS tasks. We show that some of the M&S transformations originally devised for constructing abstraction heuristics can also be used for optimal and satisficing reformulation. To do so, we provide algorithms that transform solutions for the reformulated task into plans for the original task. We also show that our M&S reformulations dominate their counterparts based on FDR representations, i.e., a suitable combination of existing M&S transformations can always do the same (and sometimes more) simplifications to any task.

To search on the FTS representation, planning algorithms and heuristics originally devised for STRIPS or FDR tasks must be adapted. As the FTS formalism is slightly more expressive than FDR, this is similar to adapting algorithms to support (a limited form of) disjunctive preconditions and conditional effects. We adapt heuristic search methods with M&S and delete-relaxation heuristics for the FTS representation. Our experimental study shows the potential of these reformulations to reduce the state space and speed-up the search. Full proofs and additional experimental results are included in a technical report (Torralba and Sievers 2019).

Representation of Planning Tasks

A planning task is a compact representation of a TS. A *transition system* (TS) is a tuple $\Theta = \langle S, L, T, s^I, \mathcal{S}^\star \rangle$ where S is a finite set of *states*, L is a finite set of *labels* each associated with a *label cost* $c(\ell) \in \mathbb{R}_0^+$, $T \subseteq S \times L \times S$ is a set of *transitions*, $s^I \in S$ is the *initial state*, and $\mathcal{S}^\star \subseteq S$ is the set of *goal states*. We use $s \in \Theta$ to refer to states in Θ and $s \xrightarrow{\ell} t \in \Theta$ to refer to its transitions. An *s-plan* for a state s is a path from s to any $s^* \in \mathcal{S}^\star$. Its cost is the summed label costs of all labels of the path. The *perfect heuristic*, $h^*(s)$, is the cost of a cheapest s-plan. An s-plan is *optimal* iff its cost equals $h^*(s)$. A plan for Π is an s^I-plan.

An abstraction is a function α mapping states in Θ to a set of *abstract states* S^α. The *abstract state space* Θ^α is $\langle S^\alpha, L, T^\alpha, s^I_\alpha, \mathcal{S}^\star_\alpha \rangle$, where $\alpha(s) \xrightarrow{\ell} \alpha(s') \in T^\alpha$ iff $s \xrightarrow{\ell} s'$ in Θ, $s^I_\alpha = \alpha(s^I)$, and $\mathcal{S}^\star_\alpha = \{\alpha(s) \mid s \in \mathcal{S}^\star\}$.

An *FDR task* is a tuple $\Pi^\mathcal{V} = \langle \mathcal{V}, \mathcal{A}, s^\mathcal{I}, \mathcal{G} \rangle$. $\mathcal{V}$ is a finite set of *variables* v, each with a *finite domain* D_v. A *partial*

state is a function s on a subset $\mathcal{V}(s)$ of $\mathcal{V}$, so that $s(v) \in D_v$ for all $v \in \mathcal{V}(s)$; s is a *state* if $\mathcal{V}(s) = \mathcal{V}$. $s^{\mathcal{I}}$is the *initial state* and the *goal* $\mathcal{G}$ is a partial state. $\mathcal{A}$ is a finite set of *actions*. Each $a \in \mathcal{A}$ is a tuple $\langle pre_a, \mathit{eff}_a, c(a)\rangle$ where pre_a and eff_a are partial states, called its *precondition* and *effect*, and $c(a) \in \mathbb{R}_0^+$ is its *cost*. An action a is applicable in a state s if $\forall_{v \in \mathcal{V}(pre_a)} s(v) = pre_a(v)$. Applying it yields the successor state $s[\![a]\!]$ with $s[\![a]\!](v) = \mathit{eff}_a(v)$ if $v \in \mathcal{V}(\mathit{eff}_a)$ and $s[\![a]\!](v) = s(v)$ otherwise.

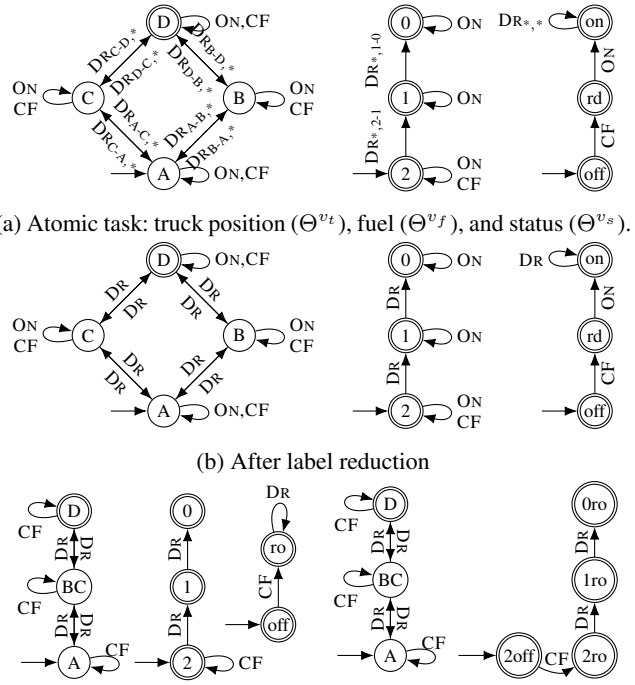
The state space of an FDR task $\Pi^{\mathcal{V}}$ is a TS $\Theta = \langle S, L, T, s^I, \mathcal{S}^\star\rangle$ where S is the set of all states, $s^I = s^{\mathcal{I}}$, $s \in \mathcal{S}^\star$ iff $\forall_{v \in \mathcal{V}(\mathcal{G})} \mathcal{G}(v) = s(v)$, $L = \mathcal{A}$, and $s \xrightarrow{a} s[\![a]\!] \in T$ if a is applicable in s.

An *FTS task* is a set of TSs $\{\Theta_1, \ldots, \Theta_n\}$ with a common set L of labels. The synchronized product $\Theta_1 \otimes \Theta_2$ of two TSs is another TS with states $S = \{(s_1, s_2) \mid s_1 \in \Theta_1 \wedge s_2 \in \Theta_2\}$, labels $L = L_1 = L_2$, transitions $T = \{(s_1, s_2) \xrightarrow{\ell} (s_1' s_2') \mid s_1 \xrightarrow{\ell} s_1' \in \Theta_1 \wedge s_2 \xrightarrow{\ell} s_2' \in \Theta_2\}$, initial state $s^{\mathcal{I}} = (s_1^{\mathcal{I}}, s_2^{\mathcal{I}})$, and goal states $\mathcal{S}^\star = \{(s_1, s_2) \mid s_1 \in \mathcal{S}_1^\star \wedge s_2 \in \mathcal{S}_2^\star\}$.

The state space of an FTS task $\Pi^{\mathcal{T}} = \{\Theta_1, \ldots, \Theta_n\}$ is defined as $\Theta = \Theta_1 \otimes \cdots \otimes \Theta_k$. Whenever it is not clear from context, we will use subscripts to differentiate states in the state space ($s, s', t \in \Theta$) and in the individual components ($s_i, s_i', t_i \in \Theta_i$). Given $s \in \Theta$, we write $s[\Theta_i]$ to refer to the projection of s onto Θ_i. A solution π for an FTS task is a sequence $s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \ldots \xrightarrow{\ell_k} s_k$ such that $s_k \in \mathcal{S}^\star$.

There is a close connection between FTS and FDR tasks, since TSs in an FTS task correspond to FDR variables with domain equal to the set of states of the TS. Then, states in FDR (which are assignments of values to variables) correspond to states in the FTS representation, which are an assignment of states s_i to each Θ_i. Given an FDR task $\Pi^{\mathcal{V}}$ it is simple to construct the corresponding FTS task, which we call the *atomic* representation of $\Pi^{\mathcal{V}}$. There is a TS Θ_v for every variable v, with one state $s_v \in \Theta_v$ per value in D_v. For every action $a \in \mathcal{A}$, there is an outgoing transition from s_v if $v \notin \mathcal{V}(pre_a)$ or $pre_a(v) = s_v$ which leads to s_v if $v \notin \mathcal{V}(\mathit{eff}_a)$ or t_v if $\mathit{eff}_a(v) = t_v$.

As running example, consider a task where a truck can drive between four locations with a limited amount of fuel and with the restriction that the engine can only be turned on with a full tank. This can be encoded as an FDR task with three variables $\mathcal{V} = \{v_t, v_f, v_s\}$ with domains $D_t = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$, $D_f = \{2, 1, 0\}$, and $D_s = \{\mathrm{off}, \mathrm{rd}, \mathrm{on}\}$ that represent the position of the truck, the amount of fuel available, and the status of the engine (off, ready, on), respectively. In the atomic FTS task, shown in Fig. 1a, there are hence three TSs $\Theta^{v_t}, \Theta^{v_f}, \Theta^{v_s}$, one for each variable. The task has an action $\textsc{Dr}_{x\text{-}y,f_1\text{-}f_2}$ with precondition $\{v_t = x, v_f = f_1, v_s = \mathrm{on}\}$ and effect $\{v_t = y, v_f = f_2\}$ for every pair of connected locations (x, y), and every $f_1, f_2 \in D_f$ s.t. $f_2 = f_1 - 1$. These actions induce transitions from x to y in Θ^{v_t}, from f_1 to f_2 in Θ^{v_f}, and a self-looping transition at state on in Θ^{v_s}. Furthermore, there exist actions check-fuel, CF, with precondition $\{v_f = 2, v_s = \mathrm{off}\}$ and effect $\{v_s = \mathrm{rd}\}$ and On with precondition $\{v_s = \mathrm{rd}\}$ and effect $\{v_s = \mathrm{on}\}$. All actions have unit cost. The initial state of the FDR task is $s^{\mathcal{I}} = \{v_t = \mathrm{A}, v_f = 2, v_s = \mathrm{off}\}$ and its goal



(a) Atomic task: truck position (Θ^{v_t}), fuel (Θ^{v_f}), and status (Θ^{v_s}).

(b) After label reduction

(c) After shrinking and removing irrelevant label On.

(d) After merging and pruning unreachable states.

Figure 1: Example FTS task where a truck must drive from A to D with a fuel capacity of 2 and the restriction to first check the fuel capacity and turn on the engine. Transitions with wildcards (*) have multiple labels, e.g., $\textsc{Dr}_{\text{A-B},*}$ stands for $\textsc{Dr}_{\text{A-B,2-1}}$ and $\textsc{Dr}_{\text{A-B,1-0}}$. Each subfigure corresponds to a reformulation (see Section).

is $\mathcal{G} = \{v_t = \mathrm{D}\}$, which translates to $(\mathrm{A}, 2, \mathrm{off})$ being the initial state (marked with incoming arrows) and all $(\mathrm{D}, *, *)$ being goal states (marked with double circles) of the FTS task.

The reverse transformation from an FTS to an FDR task is not as straightforward and it may require to introduce more FDR actions than there are labels in the FTS task. The reason is that transitions in the individual TSs are more expressive than the precondition-effect tuple of FDR actions because they can encode a limited form of angelic non-determinism, disjunctive preconditions, and conditional effects. Consider the task shown in Fig. 1b, an FTS task of the same planning task that uses label Dr for all drive actions. Translating this task to FDR requires re-introducing multiple actions to represent Dr for different pairs of locations and amounts of fuel. One reason is the non-determinism where there are multiple transitions with the same label and source state, but different targets. For example, in the state $(\mathrm{A}, 2, \mathrm{on})$, we can apply two transitions with label Dr to reach either $(\mathrm{C}, 1, \mathrm{on})$ or $(\mathrm{B}, 1, \mathrm{on})$. The non-determinism is angelic because the result is chosen by the planner at will. Also, the transitions in Θ^{v_f} encode a disjunctive precondition (Dr is applicable for $v_f = 2$ or $v_f = 1$) and conditional effects (the result of Dr is $v_f = 1$ iff $v_f = 2$ holds in the source state).

M&S Task Reformulation Framework

A task reformulation is a transformation of a task such that any solution for the new task can be transformed into a solution for the original task. We follow the definition of task reduction introduced by Tozicka et al. (2016) but without requiring the reformulated task to be smaller than the input task. Most of the reformulations we consider aim to reduce the size of the task, but reformulations that make the task bigger may be useful as well, e.g., if it makes the search space smaller.

Definition 1 (Task reformulation)**.** *A task reformulation ρ is a partial function from tasks to tasks s.t.:*

1. *$\rho(\Pi)$ is solvable if and only if Π is solvable, and*
2. *there exists a plan reconstruction function $\overleftarrow{\rho}$ that maps each solution π of $\rho(\Pi)$ to a solution $\overleftarrow{\rho}(\pi)$ of Π.*

A task reformulation is *polynomial* if both ρ and $\overleftarrow{\rho}$ can be computed in polynomial time in the size of the input task and the reconstructed plan. It is *optimal* if, given an optimal plan π of $\rho(\Pi)$, $\overleftarrow{\rho}(\pi)$ is an optimal plan of Π. We are interested in polynomial reformulations for optimal and satisficing planning. Note that we explicitly allow the reformulated plan to be exponentially larger than the input task. This is necessary for domains (e.g. Towers of Hanoi) where the original plan is exponentially long, but a reformulation with a solution that implicitly encodes the plan can be found in polynomial time.

Merge-and-Shrink Transformations

There are multiple M&S transformations that can be used to reformulate an FTS task $\Pi^{\mathcal{T}} = \{\Theta_1, \ldots, \Theta_n\}$ with labels L. A transformation is *exact* if it preserves the set of solutions and hence is an optimal reformulation.

Label reduction reduces the set of labels by mapping some of them to a common new one (Sievers, Wehrle, and Helmert 2014). It is exact if for any pair of labels $\ell, \ell' \in L$ reduced to the same label, $c(\ell) = c(\ell')$ and ℓ and ℓ' induce the same transitions in all but (at most) one Θ_i, $1 \leq i \leq n$. The task of Fig. 1b is the result of repeatedly applying exact label reduction on the atomic task of Fig. 1a. By itself, it does not affect the search space, but it reduces the amount of labels increasing the efficiency and effectiveness of other transformations.

Shrinking consists of replacing one TS $\Theta_i \in \Pi^{\mathcal{T}}$ by an abstraction thereof. This results in an abstraction of the original task, possibly introducing spurious plans that do not have any counterpart in the original task. Therefore, not all shrink transformations are suitable for task reformulation. However, using refinable abstraction hierarchies is a long standing idea in planning (Sacerdoti 1974; Bacchus and Yang 1994; Knoblock 1994). We compute refinable abstractions via shrinking strategies based on bisimulation (Milner 1971).

Definition 2 (Bisimulation)**.** *Let $\Theta = \langle S, L, T, s^I, \mathcal{S}^\star \rangle$ be a TS. An equivalence relation $\sim$ on S is a goal-respecting bisimulation iff $s \sim t$ implies that (a) $s \in \mathcal{S}^\star \leftrightarrow t \in \mathcal{S}^\star$, and (b) $\{[s'] \mid s \xrightarrow{\ell} s' \in T\} = \{[t'] \mid t \xrightarrow{\ell} t' \in T\}$ for all $\ell \in L$ where $[s]$ denotes the equivalence class of s.*

Bisimulation shrinking aggregates all states in the same equivalence class of the coarsest bisimulation of some $\Theta_i \in \Pi^{\mathcal{T}}$. This is a symmetry-reduction technique that preserves all plans and as such is an exact transformation (Helmert et al. 2014; Sievers et al. 2015). In our example (cf. Fig. 1b), states B and C of Θ^{v_t} are bisimilar in Θ^{v_t} and are hence combined into a new state BC by bisimulation shrinking (cf. Fig. 1c). Note that shrinking B and C is only possible after label reduction, since otherwise their outgoing labels differ.

When preserving optimality is not necessary, it suffices to guarantee that any abstract plan can be refined into a real plan. Hoffmann, Kissmann, and Torralba (2014) used shrinking strategies with this property for proving unsolvability with M&S. We re-define these strategies using a different nomenclature based on the notion of weak bisimulation (Milner 1971; 1990). The key idea is to consider τ-*labels* which are "internal" to a TS in the sense that they can always be taken in Θ_i without changing other TSs. The set of τ-labels for Θ_i consists of those labels ℓ having a transition $s_j \xrightarrow{\ell} s_j \; \forall s_j \in \Theta_j \; \forall \Theta_j, j \neq i$. Other definitions are possible; ours is more general than that of own-labels used by Hoffmann, Kissmann, and Torralba (2014), whereas there are stronger notions based on dominance (Torralba 2017; 2018). We use $\stackrel{\tau}{\Longrightarrow}$ to denote a (possibly empty) path using only τ-labels, and $s \stackrel{\ell}{\Longrightarrow} s'$ as a shorthand for $s \stackrel{\tau}{\Longrightarrow} \xrightarrow{\ell} \stackrel{\tau}{\Longrightarrow} s'$.

Following the observation by Haslum (2007) that it suffices to focus on paths with labels that either are outside relevant (i.e., have some effect on other variables) or reach the goal, we devise a variant of weak bisimulation that ignores some irrelevant paths. We say that a label ℓ is outside relevant for a transition system Θ_i if there exists some Θ_j with $i \neq j$ such that $s_j \xrightarrow{\ell} t_j$ for some $s_j \neq t_j$. A path $s_i \stackrel{\ell}{\Longrightarrow} s_i'$ is *relevant* for Θ_i if ℓ is outside relevant for Θ_i, or there does not exist $s_i \stackrel{\tau}{\Longrightarrow} s_i''$ such that $s_i' \sim s_i''$. Otherwise, it is safe to ignore such path in weak bisimulation because the alternative τ-path can always be used to reach s'' instead.

Definition 3 (Weak Bisimulation)**.** *Let Θ be a TS with a set τ of τ-labels, and a set T_{rel} of relevant paths. An equivalence relation $\sim$ on S is a* goal-respecting weak bisimulation *iff $s \sim t$ implies $(\exists_{s' \in \mathcal{S}^\star} s \stackrel{\tau}{\Longrightarrow} s') \leftrightarrow (\exists_{t' \in \mathcal{S}^\star} t \stackrel{\tau}{\Longrightarrow} t')$, and $\forall_{\ell \in L} \{[s'] \mid s \stackrel{\ell}{\Longrightarrow} s' \in T_{rel}\} = \{[t'] \mid t \stackrel{\ell}{\Longrightarrow} t' \in T_{rel}\}$.*

Weak bisimulation shrinking maps all weakly bisimilar states into the same abstract state. In our example (cf. Fig. 1b), ON is a τ-label in Θ^{v_s}, therefore states rd and on of Θ^{v_s} are weakly bisimilar (both have a single relevant path $\stackrel{\text{DR}}{\Longrightarrow}$ [on]) resulting in Θ^{v_s} as shown in Fig. 1c.

Another useful abstraction transformation consists of removing TSs with a *core* state. We say that a state s^C is a *core* for Θ_i if (1) for every outside relevant label ℓ there exists $s^C \stackrel{\ell}{\Longrightarrow} s^C$, (2) there is a τ-path from the initial state to s^C, and (3) there is a τ-path from s^C to a goal state. Such a TS can be abstracted away because all outside relevant labels can always be reached via a τ-path through s^C.

Merging replaces two TSs by their synchronized product. Fig. 1d shows the FTS task that results from merging Θ^{v_f}

and Θ^{v_s} of the FTS task shown in Fig. 1c. Merging is an exact transformation (Helmert et al. 2014), which comes at the price that the size of the task grows quadratically with every merge, so it increases exponentially with the number of merges. In practice, we limit the maximum size of any TS in the reformulated task, forbidding any merge that goes beyond this limit. As label reduction, by itself merging does not change the reachable search space. However, it often enables additional label reduction, shrinking, and/or pruning. In our example of Fig. 1d, CF has become a τ-label, so 2off and 2ro could be reduced by weak bisimulation.

Finally, there are multiple *pruning* techniques defined in the M&S framework. If a state s_i is *unreachable* (from the initial state) or *irrelevant* (cannot reach a goal) in any Θ_i, it can be pruned (Helmert et al. 2014). If a label ℓ is *dead* (i.e., there is no transition labeled with ℓ in any $\Theta_i \in \Pi^{\mathcal{T}}$) or *irrelevant* (i.e., all transitions labeled with ℓ are self-loop transitions), then it can be pruned (Sievers, Wehrle, and Helmert 2014). If a TS $\Theta_i \in \Pi^{\mathcal{T}}$ is the only one with a goal defined, i.e., there are no non-goal states in $\Theta_j \in \Pi^{\mathcal{T}}$ with $j \neq i$, all outgoing transitions from goal states in Θ_i can be removed (Hoffmann, Kissmann, and Torralba 2014). If a TS has only one state and no dead labels, it can be pruned. All these pruning techniques preserve at least one optimal plan and are therefore exact transformations.

Plan Reconstruction

M&S iteratively applies the transformations described above on a task $\Pi^{\mathcal{T}} = \{\Theta_1, \ldots, \Theta_k\}$, resulting in a sequence of reformulation steps $\rho_1, \ldots, \rho_n$ producing a sequence of planning tasks $\Pi^{\mathcal{T}}_0, \ldots, \Pi^{\mathcal{T}}_n$ where $\Pi^{\mathcal{T}}_0 = \Pi^{\mathcal{T}}$, and $\Pi^{\mathcal{T}}_i = \rho_i(\Pi^{\mathcal{T}}_{i-1})$ for $i \in [1, n]$. We can run any planning algorithm to find a plan $\pi^{\rho_n} = s_1^{\rho_n} \xrightarrow{\ell_1^{\rho_n}} s_2^{\rho_n} \xrightarrow{\ell_2^{\rho_n}} s_3, \ldots$ of the final task $\Pi^{\mathcal{T}}_n$. The plan reconstruction procedure is then tasked to compute a plan $\pi = s_1 \xrightarrow{\ell_1} s_2 \xrightarrow{\ell_2} \ldots$ for the original task $\Pi^{\mathcal{T}}$ from π^{ρ_n} and the sequence of reformulations.

Performing a reconstruction $\overleftarrow{\rho_i}$ for each step ρ_i has some overhead because it requires to store each intermediate task. We avoid this by combining sequences of reformulations that correspond to merge, label reduction, and bisimulation transformations. Pruning-based transformations can be ignored by the plan reconstruction procedure because the plan found is still valid for the original task without any modifications. Plan reconstruction can be done for the entire transformation at once without storing information about the intermediate planning tasks. Therefore, we have a sequence of transformations $\Pi^{\mathcal{T}} \xrightarrow{\rho_1} \Pi^{\mathcal{T}}_1 \xrightarrow{\rho_2} \Pi^{\mathcal{T}}_2 \ldots$ with only two types of reformulations to consider: merging + label reduction + bisimulation shrinking (ρ^{MLB}), and weak bisimulation shrinking ($\rho^{\tau B}$).

We first consider the reconstruction of a reformulation ρ^{MLB} on a task $\Pi^{\mathcal{T}}_i$, resulting in a task $\Pi^{\mathcal{T}}_{i+1}$. The state space of $\Pi^{\mathcal{T}}_{i+1}$ is a bisimulation of the state space of $\Pi^{\mathcal{T}}_i$, so any sequence $s_1 \xrightarrow{\ell_1} s_2 \xrightarrow{\ell_2} \ldots$ in $\Pi^{\mathcal{T}}_i$ has its counterpart $\alpha(s) \xrightarrow{\ell'_1} \alpha(s_2) \xrightarrow{\ell'_2} \ldots$ in $\Pi^{\mathcal{T}}_{i+1}$ and vice versa. To reconstruct the plan, we need two functions α and λ, mapping states and labels in $\Pi^{\mathcal{T}}_i$ to states and labels in $\Pi^{\mathcal{T}}_{i+1}$. The α function is computed by M&S heuristics and compactly represented with the so-called cascading tables or merge-and-shrink representation (Helmert et al. 2014; Helmert, Röger, and Sievers 2015). The label mapping is simply the composition of all label reduction transformations used by ρ^{MLB}.

The plan can be reconstructed step by step, starting from $s^{\mathcal{I}}$. Given the current factored state s and a step in the abstract plan $\alpha(s) \xrightarrow{\ell'} t'$, find a transition $s \xrightarrow{\ell} t$ such that $\alpha(t) = t'$ and $\lambda(\ell) = \ell'$. Note that the straightforward approach of enumerating all transitions applicable from s is not guaranteed to terminate in polynomial time because, unlike in FDR tasks where the number of successors is bounded by the number of actions, in FTS there may be exponentially many successors in the size of the task. However, one can use the cascading tables representation to retrieve a factored state $t = (t_1, \ldots, t_n)$ such that $s \xrightarrow{\ell} t$, $\ell' = \lambda(\ell)$ and $\alpha(t) = t'$. This works as follows: First, for each transition system Θ_i, obtain the set S'_i of target states t_i such that $s_i \xrightarrow{\ell} t_i$ for any label ℓ such that $\ell' = \lambda(\ell)$. Then, traverse the cascading tables and, for each intermediate table that maps states of two transition systems Θ_i, Θ_j to an abstract TS $\Theta_\gamma = \gamma(\Theta_i \otimes \Theta_j)$, compute the set of abstract states $S_\gamma = \{s_\gamma \mid \exists_{s_i \in S'_i, s_j \in S'_j}\ s_\gamma = \gamma((s_i, s_j))\}$, mapping each $s_\gamma \in S_\gamma$ to one such (s_i, s_j) pair. This allows us to keep track of one factored state for each abstract state. After all cascading tables have been traversed, it suffices to return the factored state t associated with the abstract state t'.

Proposition 1. *Label reduction, merging (up to a size limit), pruning and bisimulation shrinking are optimal and polynomial reformulations.*

Proof. It is well-known that all these techniques can be computed in polynomial time (Helmert et al. 2014; Sievers, Wehrle, and Helmert 2014). Each step of the plan can be reconstructed by traversing the cascading-tables representation, which is polynomial in the size of the input task. □

We now consider the reconstruction of a reformulation $\rho^{\tau B}$ on a task $\Pi^{\mathcal{T}}_i = \{\Theta_1, \ldots, \Theta_k\}$ where $\rho^{\tau B}$ applies weak bisimulation shrinking to some TS in $\Pi^{\mathcal{T}}_i$. We assume WLOG that Θ_1 is the shrunk TS, so $\Pi^{\mathcal{T}}_{i+1} = \{\alpha^{\tau B}(\Theta_1), \Theta_2, \ldots, \Theta_k\}$. As $\alpha^{\tau B}$ is induced by a weak bisimulation on the states of Θ_1, then for any state s in $\Pi^{\mathcal{T}}_i$ and any transition $\rho^{\tau B}(s) \xrightarrow{\ell} t^\rho$ in the reformulated task, there exists a path $s \xRightarrow{\ell} t$ in the original task such that $\rho^{\tau B}(t) = t^\rho$. Therefore, to reconstruct the plan for $\Pi^{\mathcal{T}}_i$ from a plan for $\Pi^{\mathcal{T}}_{i+1}$ one must re-introduce the τ-label transitions until reaching a state where ℓ is applicable and this results in some t such that $\rho^{\tau B}(t) \xRightarrow{\tau} t^\rho$. The search can be done locally in Θ_1 because τ-labels have self-loop transitions in other TSs. To do so, we first look for all states u_1 such that $u_1 \xrightarrow{\ell} (\xrightarrow{\tau})^* t_1$ in Θ_1 and $(\alpha^{\tau B}(t_1), s[\Theta_2], \ldots, s[\Theta_n]) = t^\rho$. Then, we run uniform-cost search from $s[\Theta_1]$ using only transitions with τ-labels until we reach such an u_1. Note that this runs in polynomial time in the size of the input task.

This procedure has similarities with red-black plan repair (Domshlak, Hoffmann, and Katz 2015), the plan reconstruction of the merge values reformulation (Tozicka et

al. 2016), or decoupled search (Gnad and Hoffmann 2018). These algorithms repair an abstract/relaxed plan by introducing additional actions to enable the preconditions ignored by the relaxed plan. Our case is slightly more complex because the same label may have multiple targets so one must ensure the remaining abstract plan is applicable in the resulting state.

If a TS Θ_i with a core state s^C was abstracted away, its corresponding path must be reconstructed as well. For each transition in the abstract plan with label ℓ, we find the shortest $s_i \overset{\ell}{\Longrightarrow} s'_i$ from the current state s_i (initialized to the initial state of Θ_i in the first iteration, and to the final state in the path of the previous iterations afterwards), and $s'_i \overset{\tau}{\Longrightarrow} s^C$. Note that to keep the plan shorter, we do not enforce the τ-path to go via the core state, but rather the condition above suffices to ensure that the rest of the plan can be reconstructed.

Proposition 2. *Weak bisimulation shrinking is a polynomial reformulation.*

Proof. The coarsest weak bisimulation of a TS can be computed by computing the bisimulation of the transitive closure of the TS over τ. Each step of the plan reconstruction corresponds to an uniform-cost search on each TS. Both operations take polynomial time in the size of the TS. □

Consider the following plan of the task shown in Fig. 1c: $(\mathrm{A}, 2, \mathrm{off}) \xrightarrow{\text{CF}} (\mathrm{A}, 2, \mathrm{ro}) \xrightarrow{\text{DR}} (\mathrm{BC}, 1, \mathrm{ro}) \xrightarrow{\text{DR}} (\mathrm{D}, 0, \mathrm{ro})$. To reconstruct the plan for the task prior to weak bisimulation shrinking (cf. Fig. 1b), we execute it and, when DR cannot be applied in rd, we insert a τ-transition with ON resulting in the plan: $(\mathrm{A}, 2, \mathrm{off}) \xrightarrow{\text{CF}} (\mathrm{A}, 2, \mathrm{rd}) \xrightarrow{\text{ON}} (\mathrm{A}, 2, \mathrm{on}) \xrightarrow{\text{DR}} (\mathrm{BC}, 1, \mathrm{on}) \xrightarrow{\text{DR}} (\mathrm{D}, 0, \mathrm{on})$. Then, we reconstruct the plan for the atomic task of Fig. 1a step by step, resulting in a plan: $(\mathrm{A}, 2, \mathrm{off}) \xrightarrow{\text{CF}} (\mathrm{A}, 2, \mathrm{rd}) \xrightarrow{\text{ON}} (\mathrm{A}, 2, \mathrm{on}) \xrightarrow{\text{DR}_{\text{A-B,2-1}}} (\mathrm{B}, 1, \mathrm{on}) \xrightarrow{\text{DR}_{\text{B-D,1-0}}} (\mathrm{D}, 0, \mathrm{on})$.

Relation to FDR Reformulation Methods

The M&S reformulations are closely related to previous FDR reformulation methods like the generalize actions (Tozicka et al. 2016), fluent merging (Seipp and Helmert 2011), and abstraction-based reformulations (Helmert 2006b; Haslum 2007; Tozicka et al. 2016). To compare reformulation methods over different formalisms, we consider that a method dominates another if it can perform the same reformulations.

Definition 4 (Dominance of Reformulation Methods)**.** *An FTS task reformulation method X dominates an FDR reformulation method Y if, given an FDR task $\Pi^{\mathcal{V}}$ and a reformulation $\rho^Y \in Y$ applicable over $\Pi^{\mathcal{V}}$, there exists a reformulation $\rho^X \in X$ such that it is applicable in $atomic(\Pi^{\mathcal{V}})$ and $\rho^X(atomic(\Pi^{\mathcal{V}})) = atomic(\rho^Y(\Pi^{\mathcal{V}}))$. We say that the domination is* strict *if there exists $\rho^X \in X$ such that it is applicable in $atomic(\Pi^{\mathcal{V}})$ but there does not exist any $\rho^Y \in Y$ applicable in $\Pi^{\mathcal{V}}$ and $\rho^X(atomic(\Pi^{\mathcal{V}})) = atomic(\rho^Y(\Pi^{\mathcal{V}}))$.*

The *generalize actions* reformulation reduces the number of FDR actions by substituting two actions by a single one if they are equal except for a precondition on a binary variable. Formally, whenever there is a variable w with domain $D_w = \{x, y\}$, and two actions a_1, a_2 s.t. $\mathcal{V}(pre_{a1}) = \mathcal{V}(pre_{a2})$, $\forall v \in (\mathcal{V}(pre_{a1}) \setminus \{w\})$ $pre_{a1}(v) = pre_{a2}(v)$, $pre_{a1}(w) = x$, $pre_{a2}(w) = y$, and $\mathit{eff}_{a1} = \mathit{eff}_{a2}$. Then, a_1 and a_2 can be replaced by a' where $\mathit{eff}_{a'} = \mathit{eff}_{a1}$ and $pre_{a'}(v) = pre_{a1}(v)$ $\forall v \in (\mathcal{V}(pre_{a1}) \setminus \{w\})$.

Theorem 1. *Exact label reduction strictly dominates the generalize actions reformulation.*

Proof Sketch. If generalize actions replaces a_1 and a_2 in $\Pi^{\mathcal{V}}$ by a', then there are labels ℓ_1 and ℓ_2 in $atomic(\Pi^{\mathcal{V}})$ that correspond to a_1 and a_2 and a TS Θ_w that corresponds to w in $\Pi^{\mathcal{V}}$. As a_1 and a_2 have the same effects and preconditions on all variables except v, then ℓ_1 and ℓ_2 are equal except for Θ_w so they can be reduced. Label reduction is more general because it may result in transitions with different targets from the same state and label, which is not possible in FDR. □

Fluent merging is an FDR reformulation inspired by the merge transformation in M&S (Seipp and Helmert 2011). It replaces two variables $v_1, v_2 \in \mathcal{V}$ by their product, resulting in a variable $v_{1,2}$ with domain $D_{v_1,v_2} = D_{v_1} \times D_{v_2}$. However, adapting the FDR actions is not straightforward since they would require disjunctive preconditions. For example, if action a_1 has a precondition on v_1 but not on v_2, then the action is applicable for several values of D_{v_1,v_2} but not for all of them. Since FDR does not allow for disjunctive preconditions, multiple copies of the actions are needed to encode the preconditions and effects on the new variable. Similarly, auxiliary actions must be added to encode a disjunctive goal whenever a goal and a non-goal variable are merged. In this case, the merge transformation does not dominate fluent merging because it does not add such auxiliary labels and transitions. This is arguably an advantage since adding them is not expected to be beneficial or, otherwise, an equivalent reformulation could be defined in M&S.

The use of abstraction for task reformulation in planning has a long history (Knoblock 1994). The key idea is to solve an abstraction of the problem and then refine the abstract solution by filling the gaps. Not all abstractions are suitable for this, since they need to ensure that any solution for the abstract task can be refined into a plan for the original task. Abstractions with this property are said to be *refinable*. Abstraction reformulations were first applied in FDR by the Fast Downward planner (Helmert 2006a). Their reformulation abstracts away any root variable in the causal graph (i.e., does not have dependencies on other variables) whose free domain transition graph is strongly connected (i.e., one can always set the variable to any desired value by applying a sequence of actions). This was generalized by Haslum (2007) into the *safe variable abstraction* reformulation under the observation that (1) it suffices to consider values that are relevant for other variables (because they are precondition or effect of an action that has another variable in its effect); and (2) the goal only needs to be achieved at the end of the plan so the goal value must be free reachable from other relevant values, but it is not necessary that other values are reachable from the goal value.

Finally, one can also ignore the difference among some values of a variable without ignoring it completely: the *merge values* reformulation reduces the domain of an FDR variable by merging several values whenever they can be switched via actions without any side effects (Tozicka et al. 2016). Formally, let v be a variable with $x, y \in D_v$, and a_1 and a_2 be actions s.t. $\mathcal{V}(pre_{a1}) = \mathcal{V}(\mathit{eff}_{a1}) = \mathcal{V}(pre_{a2}) = \mathcal{V}(\mathit{eff}_{a2}) = \{v\}$, and $pre_{a1}(v) = \mathit{eff}_{a2}(v) = x$, and $pre_{a2}(v) = \mathit{eff}_{a1}(v) = y$. Then, x may be removed from D_v, replacing every occurrence of x in A, I, and G by y.

As all these methods, weak bisimulation shrinking obtains a refinable abstraction, but on the FTS representation, taking advantage of the flexibility of M&S to compute abstractions.

Theorem 2. *Removing transition systems with core states after applying weak bismulation shrinking strictly dominates the safe variable abstraction reformulation.*

Proof Sketch. If a variable is abstracted away, abstract states corresponding to the values that appear in the preconditions of outside relevant actions are all weakly bisimilar. After shrinking, the resulting abstract state is a core state. □

Theorem 3. *Weak bisimulation shrinking strictly dominates the merge values reformulation.*

Proof Sketch. If values x, y of variable v are merged, there exist ℓ_1, ℓ_2 in $atomic(\Pi^{\mathcal{V}})$ corresponding to a_1, a_2, and a TS Θ_v representing v. As v is the only variable in the preconditions and effects of a_1 and a_2, ℓ_1 and ℓ_2 are τ-labels in Θ_v. Since $x \overset{\tau}{\Longrightarrow} y$ and $y \overset{\tau}{\Longrightarrow} x$, x and y are weakly bisimilar. □

Search on the FTS Representation

To use our reformulation framework, planning algorithms must be used to find a solution to the reformulated FTS task. Heuristic search is a leading approach for solving classical planning problems (Bonet and Geffner 2001). A compilation into an FDR task having an action for each combination of transitions with the same label in different TSs is possible, but may incur a big overhead, potentially losing any gains obtained by the reformulation methods. Here, we consider how to apply heuristic search algorithms to FTS tasks by defining the successor generation and heuristic evaluation.

Successor generation is the operation that, given a state s, generates all transitions $s \xrightarrow{l} t$ in the state space of the task. This typically is done in two steps: (1) generate the set of actions that are applicable in s and (2) for each such action obtain the corresponding successor state.

Since the number of actions in FDR tasks may be very large, iterating over all of them to check whether they are applicable in s is inefficient. The Fast Downward Planning System uses a tree data-structure to efficiently retrieve the applicable actions in a given state (Helmert 2006b). However, this data-structure relies on actions being applicable either only for one value of each variable if $v \in \mathcal{V}(pre_a)$ or in all values of such variable otherwise. This is no longer true for labels in the FTS representation. A label is applicable in a factored state s if there exists an outgoing transition $s[\Theta_i] \xrightarrow{l} t_i$ for each $\Theta_i \in \Pi^{\mathcal{T}}$. Since there may be any number of transitions in each Θ_i from any number of source states, labels may be applicable for arbitrary sets of states. We pre-compute for every abstract state $s_i \in \Theta_i$ the set of labels with an outgoing transition from s_i, denoted L_{s_i}. Then, given a state s, the set of applicable labels can be computed as $\bigcap_{\Theta_i} L_{s[\Theta_i]}$.

Step (2) is simple in FDR since the new state is a copy of s, overriding the value of variables in the effect. In the FTS representation, however, there may be multiple successors from s with label ℓ. We enumerate all possible successors by considering all outgoing transitions from $s[\Theta_i]$ in every Θ_i. To do this efficiently, for each label ℓ we divide the set of TSs in $\Pi^{\mathcal{T}}$ in three sets: the irrelevant TSs where ℓ only induces self-loop transitions, deterministic TSs where for every $s_i \in \Theta_i$ there is a single outgoing transition with ℓ, and non-deterministic TSs where there may be multiple transitions from the same source state. Only the latter require to enumerate all possible transitions, whereas irrelevant TSs are ignored and the effect on deterministic TSs can be set as in FDR tasks.

We now discuss how to derive heuristic functions for the FTS representation, which are essential to guide the search and find solutions to large tasks. As most heuristic functions have originally been defined for STRIPS or FDR, they need to be adapted to use them in FTS tasks. This is similar to adding support for a limited form of disjunctive preconditions and conditional effects. In optimal planning, we use merge-and-shrink heuristics since they are already based on FTS.

To apply our reformulation framework on satisficing planning, we adapt the FF heuristic (Hoffmann and Nebel 2001). FF is based on the delete-relaxation, ignoring the delete effects of STRIPS actions. In FDR, "ignoring deletes" is interpreted as ignoring the negative effect of the actions, so that variables accumulate values instead of replacing them. This is easily extrapolated to the FTS representation by considering that each TS may simultaneously be in multiple states.

To compute the heuristic, we compile our task into an FDR task with one unary action a_{s_i,ℓ,t_i} for each transition $s_i \xrightarrow{\ell} t_i$ in some Θ_i. This action has t_i as effect, and s_i as precondition plus additional preconditions for each other Θ_j where l is not applicable in all states. If there is a single state $s_j \in \Theta_j$ where ℓ is applicable, we add s_j to the precondition of a_{s_i,ℓ,t_i}. If there are more than one, we add an auxiliary fact to our task $f_{\ell,j}$ that represents the disjunction of those states, as well as auxiliary unary actions from each of those states to $f_{\ell,j}$.

Afterwards, we retrieve the relaxed plan as a set of transitions $s_i \xrightarrow{\ell} t_i$, and add the cost of all their labels to obtain the heuristic value. One difference to FF for FDR is that there, FF counts each action only once because no action needs to be applied more than once in delete-free tasks. We do not do this to avoid underestimating the goal distance when the same label may have different effects (e.g. label DR in Fig.1b).

The delete-relaxation is also useful to select preferred actions. In FDR, an action is preferred in state s if it belongs to the relaxed plan of FF for s. In FTS, we consider $s \xrightarrow{\ell} t$ to be preferred if the relaxed plan from s contains a transition

labeled with ℓ and with target t_i s.t. $\exists \Theta_i t[\Theta_i] = t_i$.

Experiments

We implemented the M&S reformulation framework in Fast Downward (FD) (Helmert 2006b), using its existing M&S framework (Sievers 2018) and extending it with weak bisimulation as well as pruning transformations that remove dead labels and irrelevant TSs and labels. We also modified the layout of the algorithm: firstly, since our pruning transformations might trigger further pruning opportunities, we always repeatedly apply them until a fixpoint is reached. Secondly, we run label reduction and shrinking on the atomic FTS task until no more simplifications are possible. Finally, we cannot exactly control the amount of shrinking done because this would result in non-refinable abstractions that do not admit plan reconstruction. Instead, we restrict merging to satisfy the size limit and only shrink after merging and pruning.

To consider the effects of some of the M&S transformations on the task reformulation individually, we consider the following configurations. As the simplest baseline, we only transform the FDR task (FDR) into the atomic FTS task (a), without any further transformations. This does not affect the state space at all, but serves for quantifying the overhead of our implementation over FD, mainly due to using different data structures to represent the task and perform successor generation. Another variant of atomic adds exact label reduction and shrinking (a-ls), either based on bisimulation for optimal planning or weak bisimulation for satisficing planning. Other configurations combine label reduction and shrinking with a merge strategy. For the latter, we consider DFP (d-ls) and sbMIASM (m-ls, called dyn-MIASM originally) (Sievers, Wehrle, and Helmert 2014), with a size limit of 1000 on the resulting product. We did not find qualitative differences with size limits of 100 and 10000. We impose a time limit of $900s$ on the reformulation process. For the overall planning, we use a limit of 3.5 GiB and $1800s$. We use all STRIPS benchmarks from the optimal/satisficing tracks of all IPCs, two sets consisting of 1827/1816 tasks across 48 unique domains.[1]

Search Space Reduction

To assess the impact of our task reformulations on the reachable state space, we run uniform-cost search and evaluate the number of expansions until the last f-layer. Fig. 2 compares the FDR representation against a-ls and d-ls with bisimulation (top) and weak bisimulation (bottom) shrinking. We observe that even with only label reduction and bisimulation shrinking (a-ls) there are state space reductions of up to one order of magnitude in some cases. Most of these gains are due to shrinking, given that label reduction does not change the state space and pruning cannot be performed often in the atomic representation due to the preprocessing of FD. When using merge transformations (d-ls), state space reductions can often be of up to several orders of magnitude. It is worth noting that merging does not affect the state space, so this reduction is due to the synergy with pruning and shrinking.

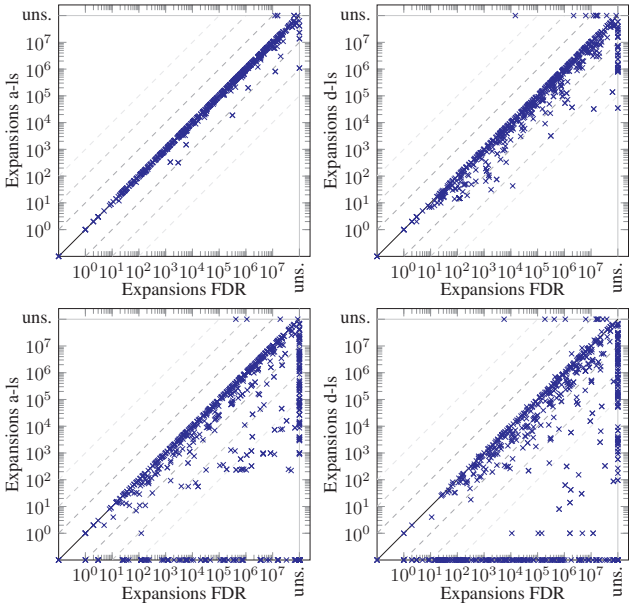


Figure 2: Expansions until last f-layer of blind search on the FDR task and different reformulated FTS tasks, using bisimulation (top) and weak bisimulation (bottom) for shrinking.

If optimality does not need to be preserved, larger reductions can be achieved with weak bisimulation shrinking. In this case, 305 tasks (including entire domains like logistics, miconic, movie, rovers, and zenotravel) can be solved during the reformulation resulting in 0 expansions (points on the x-axis). The reason is that weak bisimulation shrinks away entire TSs (e.g., if they form a single connected component with actions without side preconditions or effects, which translate to τ-labels). An example is logistics: as trucks/airplanes can always freely change their location with the drive/fly action, weak bisimulation simplifies the TSs describing their position, after which the TSs for packages can also be simplified. Previous abstraction reformulation approaches solved many of these domains too, with the exception of Rovers, where they obtained reductions but without completely simplifying the domain. With merge reformulations, 460 tasks are solved with DFP (completely solving transport-opt), and 514 with MIASM (solving all but two instances in parcprinter-opt). This is remarkable given the low limit of 1000 abstract states.

Results with Informed Search

We evaluate the impact of our reformulations in terms of coverage (see Table 1), expansions, and total time (see Fig. 3). On the optimal benchmarks, we run A* with $h^{\max}$ and M&S with DFP using a 50000 size limit and (approximate) bisimulation shrinking. On the satisficing benchmarks, we run lazy greedy search with h^{FF}, with and without preferred operators. The comparison of FDR and atomic (a) shows that our implementation has some overhead. Both

[1] Implementation: `https://doi.org/10.5281/zenodo.3232878`, dataset with benchmarks: `https://doi.org/10.5281/zenodo.3232844`.

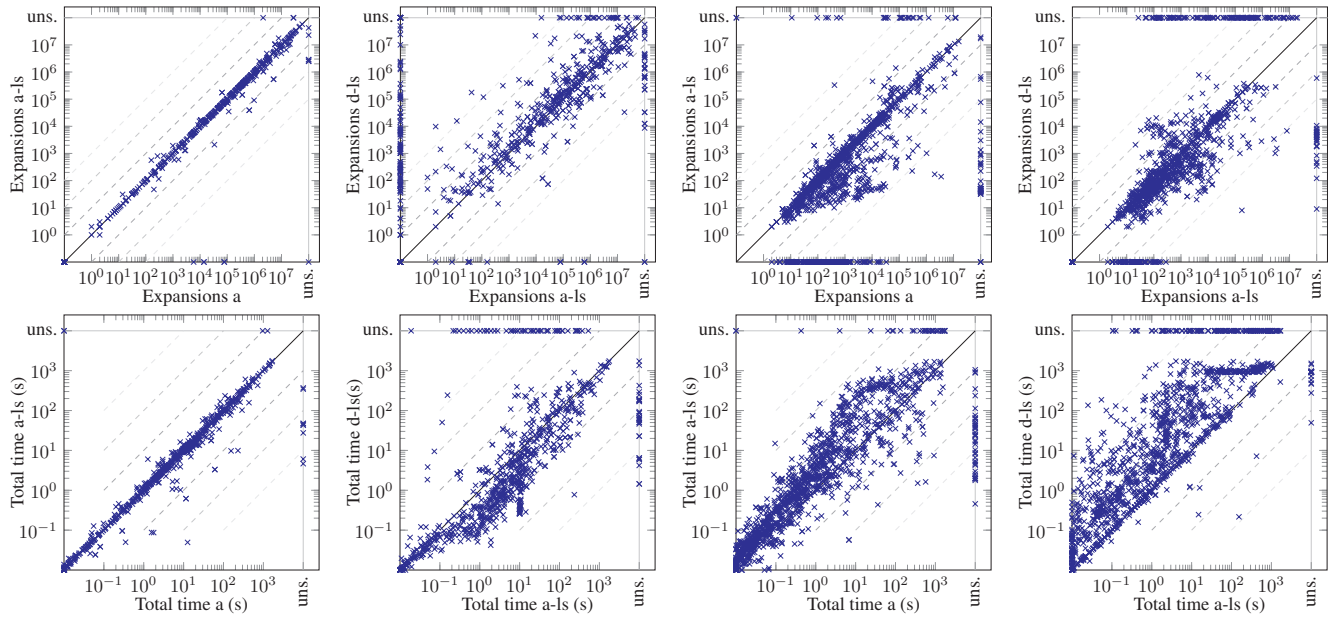


Figure 3: Expansions until last f-layer and total time of a vs. a-ls (left) and a-ls vs. d-ls (right) for A*with M&S (left block) and lazy greedy search with $h^{\text{FF}}$and preferred operators (right block).

configurations explore the same state space with very similar heuristics. $h^{\max}$ and h^{FF} are computed in the same way with no big overhead and the runtime of plan reconstruction is usually negligible. In terms of heuristic value, $h^{\max}$ is identical and h^{FF} only differs due to tie-breaking and because some actions may be counted twice. One of the main sources of overhead is the memory used to represent FTS tasks. Our data structures use $O(|L|)$ memory on each TS, whereas in FDR no memory is wasted for variables not mentioned in the preconditions or effects.

Label reduction and shrinking on the atomic FTS task (a-ls) is useful in most cases, increasing total coverage in all configurations. This reformulation reduces the state space as well as the task description size (i.e. reducing the TSs in the FTS representation). Therefore, gains in expanded nodes usually translate into lower search times, and it can pay off despite the overhead of the precomputation phase on total time.

Merge reductions (d-ls), however, are oftentimes harmful in combination with delete-relaxation heuristics ($h^{\max}$ and h^{FF}), due to the overhead caused by increasing the task size. Nevertheless, they can be very useful in some domains, whenever there is enough synergy with pruning (e.g. woodworking, tpp) or shrinking (e.g. childsnack). Indeed, for all heuristics we tried, merge reformulations are useful in at least a few domains. This is also reflected in the orcl column that shows how many instances are solved by any of our configurations. This is often much larger than our atomic configuration, but also than the FDR baseline, showing that if the right reformulations are chosen for each domain, they can compensate for the overhead of using an FTS representation.

The results of d-ls with M&S heuristics are different be-

	FDR	a	a-ls	d-ls	m-ls	tot	orcl
FDR	–	**12**	**13**	**37**	**36**	797	
a	1	–	1	**36**	**36**	770	
a-ls	3	**4**	–	**36**	**35**	780	
d-ls	2	2	1	–	7	600	
m-ls	4	4	4	**19**	–	632	
FDR	–	2	3	12	14	822	
a	**4**	–	1	**13**	**16**	826	
a-ls	**7**	**4**	–	**13**	**16**	831	
d-ls	**13**	11	10	–	11	815	
m-ls	**16**	15	15	**16**	–	849	

$h^{\max}$: 801
$h^{\text{M\&S}}$d: 910

	FDR	a	a-ls	d-ls	m-ls	tot	orcl
FDR	–	**18**	15	**27**	**22**	1326	
a	6	–	13	**28**	**22**	1272	
a-ls	**18**	**15**	–	**31**	**24**	1368	
d-ls	10	10	4	–	11	1208	
m-ls	13	15	7	**21**	–	1224	
FDR	–	**17**	**15**	**24**	**23**	1502	
a	8	–	**11**	**25**	**24**	1461	
a-ls	13	8	–	**26**	**26**	1471	
d-ls	9	6	2	–	15	1357	
m-ls	9	7	3	**16**	–	1322	

h^{FF}: 1413
h^{FF}p.: 1589

Table 1: Domain comparison of coverage for A*(left) with $h^{\max}$(top) and M&S (bottom), and lazy greedy search (right) with h^{FF}, without (top) and with (bottom) preferred operators. A value in row x and column y denotes the number of domains where x is better than y. It is bold if this is higher than the value in y/x. Column "tot" shows total coverage and "orcl" shows the oracle, i.e., per-task maximized, coverage over our algorithms (thus excluding FDR).

cause there are more cases where the heuristic is less informed after the d-ls reformulation, increasing the number of expansions. There is also a large number of instances where the heuristic value for the initial state is perfect for a-ls whereas a large amount of search is needed with d-ls. The reason is that the options available for the merge strategy during the reformulation are reduced by the limit on abstract states, leading to different merge decisions, and possibly degrading the quality of the heuristic. However, with

M&S heuristics there is no overhead in runtime so d-ls pays off more often.

The rightmost two columns of Fig. 3 show results with $h^{\text{FF}}$and preferred operators for satisficing planning. The reductions obtained by weak bisimulation shrinking are much stronger than by optimality preserving strategies, improving the performance of a-ls and d-ls in terms of expanded nodes. In terms of runtime, a-ls is useful in many cases despite the overhead caused by spending up to 900s in preprocessing. Merge reformulations, however, increase the computational cost of the heuristic, so they do not pay off over a-ls except in a few cases where the reduction is huge.

Conclusion

In this work, we use the M&S framework for task reformulation and analyze its advantages over reformulations in FDR. Our results show a large potential of state space reductions, that sometimes can solve entire domains without any search.

The framework has even more potential by integrating new reformulation methods like subsumed transition pruning (Torralba and Kissmann 2015), or graph factorization (Wehrle, Sievers, and Helmert 2016). Our results also show that not all reformulations are always helpful. Thus, to materialize all this potential, methods to automatically select the best reformulation method for each domain are also of great interest (Gerevini, Saetti, and Vallati 2009; Fuentetaja et al. 2018).

Acknowledgments

Silvan Sievers has received funding for this work from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 817639). Álvaro Torralba has received support by the German Research Foundation (DFG), under grant HO 2169/5-1 and DFG grant 389792660 as part of TRR 248 (see `https://perspicuous-computing.science`).

References

Bacchus, F., and Yang, Q. 1994. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence* 71:43–100.

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

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

Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69(1–2):165–204.

Coles, A., and Coles, A. 2010. Completeness-preserving pruning for optimal planning. In Coelho, H.; Studer, R.; and Wooldridge, M., eds., *Proc. of ECAI'10*, 965–966. Lisbon, Portugal: IOS Press.

Domshlak, C.; Hoffmann, J.; and Katz, M. 2015. Red-black planning: A new systematic approach to partial delete relaxation. *Artificial Intelligence* 221:73–114.

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2006. Directed model checking with distance-preserving abstractions. In Valmari, A., ed., *Proc. of SPIN'06*, volume 3925 of *Lecture Notes in Computer Science*, 19–34. Springer-Verlag.

Dräger, K.; Finkbeiner, B.; and Podelski, A. 2009. Directed model checking with distance-preserving abstractions. *International Journal on Software Tools for Technology Transfer* 11(1):27–37.

Fuentetaja, R.; Barley, M. W.; Borrajo, D.; Douglas, J.; Franco, S.; and Riddle, P. J. 2018. Meta-search through the space of representations and heuristics on a problem by problem basis. In *Proc. AAAI'18*, 6169–6176.

Gerevini, A.; Saetti, A.; and Vallati, M. 2009. An automatically configurable portfolio-based planner with macro-actions: PbP. In *Proc. ICAPS'09*, 350–353.

Gnad, D., and Hoffmann, J. 2018. Star-topology decoupled state space search. *Artificial Intelligence* 257:24 – 60.

Haslum, P. 2007. Reducing accidental complexity in planning problems. In *Proc. IJCAI'07*, 1898–1903.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge & shrink abstraction: A method for generating lower bounds in factored state spaces. *Journal of the Association for Computing Machinery* 61(3).

Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In *Proc. ICAPS'07*, 176–183.

Helmert, M.; Röger, G.; and Sievers, S. 2015. On the expressive power of non-linear merge-and-shrink representations. In *Proc. ICAPS'15*, 106–114.

Helmert, M. 2006a. Fast (diagonally) downward. In *IPC 2006 planner abstracts*.

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

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

Hoffmann, J.; Kissmann, P.; and Torralba, Á. 2014. "Distance"? Who Cares? Tailoring merge-and-shrink heuristics to detect unsolvability. In *Proc. of ECAI'14*.

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

Milner, R. 1971. An algebraic definition of simulation between programs. In *Proc. IJCAI'71*, 481–489.

Milner, R. 1990. Operational and algebraic semantics of concurrent processes. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*. Elsevier and MIT Press. 1201–1242.

Sacerdoti, E. D. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5:115135.

Seipp, J., and Helmert, M. 2011. Fluent merging for classical planning problems. In *ICAPS 2011 Workshop on Knowledge Engineering for Planning and Scheduling*, 47–53.

Sievers, S.; Wehrle, M.; Helmert, M.; Shleyfman, A.; and Katz, M. 2015. Factored symmetries for merge-and-shrink abstractions. In *Proc. AAAI'15*, 3378–3385.

Sievers, S.; Wehrle, M.; and Helmert, M. 2014. Generalized label reduction for merge-and-shrink heuristics. In *Proc. AAAI'14*, 2358–2366.

Sievers, S. 2018. Merge-and-shrink heuristics for classical planning: Efficient implementation and partial abstractions. In *Proc. of SOCS'18*, 90–98.

Torralba, Á., and Kissmann, P. 2015. Focusing on what really matters: Irrelevance pruning in merge-and-shrink. In *Proc. of SOCS'15*, 122–130.

Torralba, Á., and Sievers, S. 2019. Merge-and-shrink task reformulation for classical planning: Technical report. Technical Report CS-2019-004, University of Basel.

Torralba, Á. 2017. From qualitative to quantitative dominance pruning for optimal planning. In *Proc. of IJCAI'17*, 4426–4432.

Torralba, Á. 2018. Completeness-preserving dominance techniques for satisficing planning. In *Proc. of IJCAI'18*, 4844–4851.

Tozicka, J.; Jakubuv, J.; Svatos, M.; and Komenda, A. 2016. Recursive polynomial reductions for classical planning. In *Proc. ICAPS'16*, 317–325.

Wehrle, M.; Sievers, S.; and Helmert, M. 2016. Graph-based factorization of classical planning problems. In *Proc. IJCAI'16*, 3286–3292.

Information Shaping for Enhanced Goal Recognition of Partially-Informed Agents

Sarah Keren, Haifeng Xu, Kofi Kwapong, David Parkes, Barbara Grosz
School of Engineering and Applied Sciences
Harvard University
skeren@seas.harvard.edu, hxu@seas.harvard.edu, kwapongk@college.harvard.edu, parkes@eecs.harvard.edu, grosz@eecs.harvard.edu

Abstract

We extend goal recognition design by considering a two-agent setting in which one agent, the *actor*, seeks to achieve a goal but has only partial information about its environment. The second agent, the *recognizer*, has perfect information and aims to recognize the actor's goal from its behavior as quickly as possible. As a one-time offline intervention the recognizer can selectively reveal information to the actor. The problem of selecting which information to reveal, which we call *information shaping*, is challenging because the space of information shaping options may be extremely large, and because more information revelation need not make an agent's goal easier to recognize. We formally define this problem, and suggest a pruning approach for efficiently searching the space of information shaping options. We demonstrate the ability to facilitate recognition via information shaping and the efficiency of the suggested method on a set of standard benchmarks.

Introduction

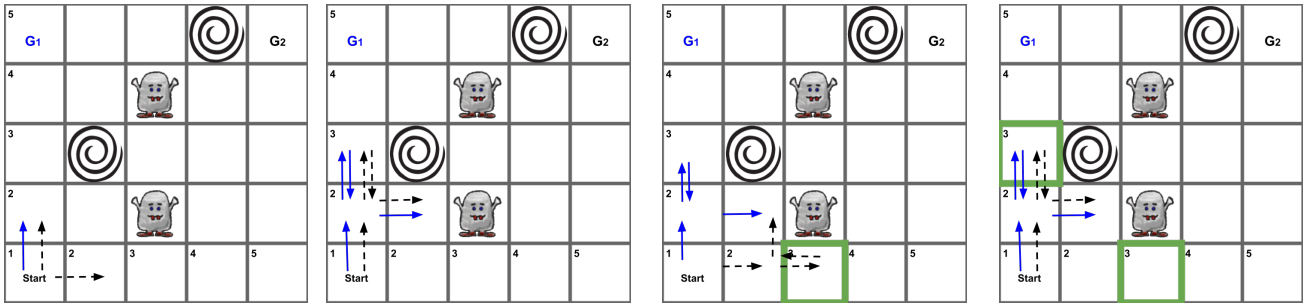
Goal recognition is the task of detecting the goal of agents by observing their behavior (Cohen, Perrault, and Allen 1981; Kautz and Allen 1986; Ramirez and Geffner 2010; Carberry 2001; Sukthankar et al. 2014). We consider a two-agent goal recognition setting, where the first agent, the *actor*, has partial information about a deterministic environment and seeks to achieve a goal. The second agent, the *recognizer*, has perfect information, and tries to deduce the actor's goal as early as possible, by analyzing the actor's behavior.

As a one time offline intervention, and with the objective of facilitating the recognition task, the recognizer can apply a limited number of *information shaping* modifications, implemented as changes to the actor's sensor model. Such modifications can help to differentiate the actor's behavior for different goals, potentially making it easier to interpret.

The ability to quickly understand what an agent is trying to achieve, without expecting it to explicitly communicate its objectives, is important in many applications. For example, in an assistive cognition setting (Kautz et al. 2003), it may be critical to know as early as possible when a visually impaired user is approaching a hot oven, giving the system time to react to the dangerous situation (e.g., by calling for help, reducing the heat, etc.). In security applications it may be important to early detect users aiming at a specific destination (Boddy et al. 2005), giving the system enough time to send human agents to further investigate potential threats. Early detection is also important in human-robot collaborative settings (Levine and Williams 2014), where a robot aims to recognize what component a human user is trying to assemble, so it can gather the tools needed for the task in a timely fashion. Common to all these settings, is that agents have *incomplete information* about their environment. This affects their behavior and is key to the ability to interpret it. In addition, these settings can be controlled and modified in various ways. Specifically, it may be possible to modify an agent's behavior by manipulating its knowledge and its need to act in order to acquire new information. Such manipulations may induce behaviors that can be quickly associated to a specific goal. To demonstrate, in an assisted cognition setting, an auditory signal can inform users about a hot oven. Early notification potentially causes users aiming at a different goal (e.g., the cupboard) to move away from the oven, supporting early recognition of dangerous situations.

This work extends the *goal recognition design* (GRD) framework, which deals with redesigning agent settings in order to facilitate early goal detection (Keren, Gal, and Karpas 2014; Wayllace et al. 2016). Until now, GRD work has assumed that agents have perfect knowledge of their environment. In this paper, we extend the framework to support agents with incomplete knowledge. Specifically, we focus on GRD in deterministic environments, and use contingent planning (Bonet and Geffner 2011; Brafman and Shani 2012a; Muise, Belle, and McIlraith 2014; Albore, Palacios, and Geffner 2009) to represent the actor. The design objective is to minimize *worst case distinctiveness* (*wcd*) (Keren, Gal, and Karpas 2014), which represents the longest sequence of actions (or path cost) that is possible before the actor's goal is recognized. Note that in some instances the goal may remain unrecognized, and even go unattained, in which case the *wcd* is simply the number of actions (or accumulated action cost) until the end of execution.

To minimize *wcd* we use *information shaping* and require that the information conveyed to the actor is truthful and cannot mislead. Specifically, we use *sensor extensions* to improve information about the value of some environment variables. This is a challenging problem because the number of possible design options may be extremely large. Also, as we demonstrate below, more information need not make an agent's goal easier to recognize.



(a) The goal recognition setting: an actor to either goal may move up.

(b) Plans executed by an actor aiming at G_1 (solid arrows) and G_2 (dashed arrows).

(c) Plans for each goal when the recognizer reveals (3,1) is safe

(d) Plans for each goal when the recognizer reveals both (3,1) and (1,3) are safe.

Figure 1: An example of a GRD-APK problem

Example 1 *As a simple example, consider Figure 1(a), depicting a variation of the Wumpus domain (Russell and Norvig 2016), where a partially informed actor has one of two goals (indicated by G_1 and G_2 in the image), and needs to achieve it without falling into pits or encountering a deadly wumpus. The actor knows its current position, but initially does not know the locations of the pits and wumpuses. When in a cell adjacent to a pit, it senses a 'breeze' and it can smell the stench of a wumpus from an adjacent cell. The recognizer has perfect information: it knows the locations of the actor, the pits (e.g., the spiral at cell $(2,3)$) and the wumpuses (e.g., cell $(3,2)$).*

The actor starts at 'Init'. With no breeze or stench, it deduces the adjacent cells are safe. *In this example, we will assume the actor is optimistic when planning but conservative when acting (Bonet and Geffner 2011). For planning, the actor makes the most convenient assumptions about (chooses the value of) unknown variables, plans accordingly, and revises the assumptions and re-plans if these assumptions are refuted during execution. If there are multiple cost-minimal plans (under optimism), we assume the actor selects one that requires making as few assumptions as possible (and arbitrarily otherwise). Consequently, an agent aiming at G_1 will start by moving up. In contrast, an uninformed agent aiming at G_2 is indifferent to going up or right, and may go either way. Because of this, moving up from the initial state leaves the goal unrecognized. Let us suppose (Figure 1b) that plans to both goals start by moving up two steps. After sensing a breeze at cell (1,3), not knowing which adjacent cells have a pit, the actor backtracks and moves right. After sensing a 'breeze' and 'stench', the actor deduces there is a wumpus at cell (3,2), and realizes that it will sense a stench at cell (3,1), without having the option of verifying that cell (4,1) is safe. With no more cells to explore, it halts at (2,2) leaving the goal unrecognized even after it terminates execution, setting* wcd *to* 4.

To promote early recognition, the recognizer can share information with the actor before it starts execution, for example by revealing safe cells. However, suppose there is a budget, limiting the number of facts that can be revealed. If the recognizer chooses to reveal cell $(3,1)$ is safe (Figure 1(c)), an actor aiming at G_2 (originally indifferent to moving up or right) prefers moving right from the initial state. In contrast, an actor aiming at G_1 still prefers moving up. The goal of the actor becomes clear as soon as the first step is performed and wcd *is minimized (*wcd*=0). Note that if, in addition, the recognizer reveals that cell $(1,3)$ is safe (Figure 1(d)), the initial situation is recovered, since an actor to either goal may now choose to move up given its beliefs about minimal plans. This illustrates the need to carefully select the information to reveal in order to facilitate the recognition task.*

The contributions of this work are fourfold. First, we extend the GRD framework to support agents with partial information. We refer to our extended setting as *GRD for Agents with Partial Knowledge* (GRD-APK), and suggest information shaping modifications that can be applied to support goal recognition. Second, since our extended design setting induces a large search space of possible information shaping modifications and since previous approaches to design do not apply to our setting, we present a novel pruning method, and specify the conditions under which it is safe, so that at least one optimal solution is not pruned. Third, we implement our suggested approach, using STRIPS (Fikes and Nilsson 1972) to represent our generic and adaptable redesign process. Finally, we evaluate the algorithm on a set of standard benchmarks, and demonstrate both *wcd* reduction achievable through information shaping and the efficiency of our approach.

Background: Planning Under Partial Observability

To support agents with partial knowledge, we follow Bonet and Geffner (2011) and consider contingent planning under partial observability, formulated as follows.

Definition 1 ***A planning under partial observability with deterministic actions (PPO-det) problem*** *is a tuple $P = \langle \mathcal{F}, \mathcal{A}, I, G, \mathcal{O} \rangle$ where $\mathcal{F}$ is a set of fluent symbols, $\mathcal{A}$ is a set of actions, I is a set of* clauses *over fluent-literals defining the initial situation, G is a set of fluent-literals defining the goal condition, and $\mathcal{O}$ represents the agent sensor model.*

An action $a \in \mathcal{A}$ is associated with a set of preconditions $prec(a)$, which is the set of fluents that need to hold for a to be applicable, and conditional effects $\mathit{eff}(a)$, which is a set

of pairs $(\mathcal{F}_{cond}, \mathcal{F}_{\mathit{eff}})$ s.t. $\mathcal{F}_{\mathit{eff}} \subseteq \mathcal{F}$ become true if $\mathcal{F}_{cond} \subseteq \mathcal{F}$ are true when a is executed.

The sensor model $\mathcal{O}$ is a set of observations $o \in \mathcal{O}$ represented as pairs (C, L) where C is a set of fluents and L is a positive fluent, indicating that the value of L is observable when C is true. Each observation $o = (C, L)$ can be conceived as a sensor on the value of L that is activated when C is true.

A state s is a truth valuation over the fluents $\mathcal{F}$ ('true' or 'false'). For an agent, the value of a fluent may be known or unknown. A fluent is *hidden* if its true value is unknown. A *belief state* b is a non-empty collection of states the agent deems possible at some point. A formula $\mathbb{F}$ *holds* in b if it holds for every state $s \in b$. An action a is *applicable* in b if the preconditions of a hold in b, and the *successor* belief state b' is the set of states that results from applying the action a to each state s in b. When an observation $o = (C, L)$ is activated, the successor belief is the *maximal* set of states in b that agree on L. The initial belief is the set of states that satisfy I, and the goal belief are those that satisfy G. A formula is *invariant* if it is true in each possible initial state, and remains true in any state that can be reached from the initial state. A *history* is a sequence of actions and beliefs $h = b_0, a_0, b_1, a_1, \ldots, b_n, a_n, b_{n+1}$. It is *complete* if the performing agent reaches a goal belief state.

A solution to a PPO-det problem P is a *policy* π, which is a partial function from beliefs to actions. A policy is *deterministic* if any belief b is mapped to at most one action. Otherwise it is *non-deterministic*. A history h **satisfies** π, if $\forall i\ 0 \leq i \leq n$, $a_i \in \pi(b_i)$. There are three types of policies: *weak*, when there is at least one complete history that satisfies the policy, *strong*, where a goal belief is guaranteed to be achieved within a fixed number of steps, and *strong cyclic*, where a goal belief is guaranteed to be achieved, but with no upper bound on the cost (length) of the solution. Our framework, suggested next, supports all three policy types.

Goal Recognition Design for Agents with Partial Knowledge (GRD-APK)

The *goal recognition design for agents with partial knowledge problem* (GRD-APK) consists of an initial goal recognition setting, a measure by which a setting is evaluated, and a design model, which specifies the information shaping modifications that can be applied. We first define each component separately.

Goal Recognition

A goal recognition setting can be defined in various ways (Sukthankar et al. 2014), but typically includes a description of the underlying environment, the way agents behave in it to achieve their goal, and the observations collected by the goal recognizing agent. Accordingly, our goal recognition model supports two agents; a partially informed contingent planning *actor* (Definition 1) with a goal, that executes history h until reaching a goal belief or halting when no action is applicable. The second agent is a perfectly informed *recognizer*, that analyzes the actor's state transitions in order to recognize the actor's goal.

Definition 2 *A **goal recognition for agents with partial knowledge problem** (GR-APK) is a tuple $R = \langle E, \mathcal{G}, \mathcal{O}^{ac}, \{\Pi(G)\}_{G\in\mathcal{G}}\rangle$ where:*

- *$E = \langle \mathcal{F}, \mathcal{A}, I\rangle$ is the environment, which consists of the fluents $\mathcal{F}$, actions $\mathcal{A}$ and initial state I as defined in Definition 1 (a cost $\mathcal{C}(a)$ for each action $a \in \mathcal{A}$ may also be specified),*
- *$\mathcal{G}$ is a set of possible goals G, s.t. $|\mathcal{G}| \geq 2$ and $G \subseteq \mathcal{F}$,*
- *$\mathcal{O}^{ac}$ is the actor's sensor model (Definition 1), and*
- *$\{\Pi(G)\}_{G\in\mathcal{G}}$ are the set of policies $\Pi(G)$ an agent aiming at goal $G \in \mathcal{G}$ may follow.*

The cost of history h, denoted $\mathcal{C}_a(h) = \Sigma_i \mathcal{C}(a_i)$, is the accumulated cost of the performed actions (equal to path length when action cost is uniform). In executing h, the actor follows a possibly non-deterministic policy π from the set $\Pi(G)$ of possible policies to its goal.

The set $\Pi(G)$ of policies to each goal is typically implicitly defined via the solver used by the actor to decide how to act in each belief state. In Example 1 we described an example of such a solver, which we will formally define in the next section. The GRD-APK framework is well defined for any solver that provides a mapping $\mathcal{B} \rightarrow 2^{\mathcal{A}}$, specifying the set of possible actions an agent may execute at each reachable belief state $b \in \mathcal{B}$ (e.g., (Bonet and Geffner 2011; Muise, Belle, and McIlraith 2014).

In our setting, the actor and recognizer both know the environment E and the set $\mathcal{G}$ of possible goals. While the partially informed actor needs to collect information about the environment via its sensor model $\mathcal{O}^{ac}$ in order to achieve its premeditated goal, the recognizer knows the true state of the world and the actor's solver and sensor, but does not know the actor's goal. The recognizer observes the actor's transitions between belief states and analyzes them in order to recognize the actor's goal.[1]

Evaluating a GR-APK model

The *worst case distinctiveness* (*wcd*) measure represents the maximum number of actions an actor can perform (in general, maximum total cost incurred by the actor) before its goal is revealed. To define *wcd* we first define the relationship between the observations collected by the recognizer when an actor follows history h, which in our case correspond to the actor's transitions between belief states, and a goal. As mentioned above, we say that a history *satisfies* a policy, if it is a possible execution of the policy. In addition, a history *satisfies* a goal, if satisfies a possible policy to the goal.

Definition 3 *Given a GR-APK model R, history h* **satisfies** *policy π in R, if $\forall i\ 0 \leq i \leq n$, $a_i \in \pi(b_i)$. In addition, h* **satisfies** *goal $G \in \mathcal{G}$ in R if $\exists \pi \in \Pi(G)$ s.t. h satisfies π.*

[1] Since we are analyzing the goal recognition setting, and need to account for all possible observations of agent behavior, we do not specify a particular history to be analyzed, which is a typical component in goal recognition models (e.g., (Ramirez and Geffner 2010; Pereira, Oren, and Meneguzzi 2017). Instead, in facilitating goal recognition via design, our model characterizes the different actor behaviors in the system, and the way they are perceived by the recognizer.

Let $\mathcal{G}^{rec}(h)$ represent the set of goals that history h satisfies, i.e., the set of goals the recognizer deems as possible actor goals. We define a history as *non-distinctive* if it satisfies more than one goal.

Definition 4 *Given a GR-APK model R, a history h is* **non distinctive** *in R, if exists $G, G' \in \mathcal{G}$ s.t. $G \neq G'$, and h satisfies G and G'. Otherwise, it is distinctive.*

We denote the set of non-distinctive histories of a GR-APK model R by $H^{nd}(R)$.

Definition 5 *The* **worst case distinctiveness** *of a model R, denoted by* $\text{wcd}(R)$ *is:*

$$\text{wcd}(R) = \begin{cases} \max_{h \in H^{nd}(R)} \mathcal{C}_a(h) & H^{nd}(R) \neq \emptyset \\ 0 & otherwise \end{cases}$$

That is, *wcd* is the maximum cost history for which the goal is not determined, or zero if there is no such history. Recall that in some instances the goal may remain unrecognized, and even go unattained, in which case the *wcd* is simply the number of actions (or accumulated action cost) until the end of execution. Also recall that a policy may be strong cyclic, potentially containing infinite loops. A policy with such a cycle is considered to have a history with infinite cost. In particular, since such a history may be non-distinctive, this means *wcd* in this setting may be infinite.

Information Shaping

Our interest here is in modulating the behavior of the actor through information shaping. By changing the actor's knowledge, we can potentially change its behavior and the way by which it acquires the information needed to achieve it's goal. We restrict the information shaping interventions to be truthful so that they cannot convey false information. In the context of contingent, partially-informed planning agents, this requirement is naturally implemented by requiring that we may only improve the actor's sensor model, i.e., improving its ability to access the value of some environment feature. We define *sensor extension* modifications, which add a single observation to a sensor model, using $\boldsymbol{\mathcal{O}}$ to denote the set of all sensor models.

Definition 6 *A modification $\delta : \boldsymbol{\mathcal{O}} \rightarrow \boldsymbol{\mathcal{O}}$ is a* **sensor extension** *if $\delta(\mathcal{O}) = \mathcal{O} \cup \{o\}$, for all $\mathcal{O} \in \boldsymbol{\mathcal{O}}$, and for some $o = (C, L)$.*

Sensor extensions correspond to adding new sensors to the environment, or, as a special case, communicating to the actor the value of a feature (setting $C = \emptyset$).

To demonstrate, in Example 1 the recognizer can allow the actor to sense a stench in cell $(1, 2)$, two (rather than one) cells away from the wumpus in cell (3,2). This extension is implemented by adding the observation $o = (C = AgentAtCell(1, 2), L = BreezeInCell(2, 2))$ to the actor's sensor model. This could be realized through a visual indication or sign, similar to the auditory signal indicating the oven is hot in the assisted cognition example. The recognizer could also directly communicate with the actor and inform it about the location of a wumpus, or reveal a location without a wumpus. (e.g., $(C = True, L = WumpusAtCell(4, 4))$.

We are now ready to define a GRD-APK problem.

Definition 7 *A* **goal recognition design for agents with partial knowledge problem** *(GRD-APK) is defined as a tuple $T = \langle R_0, \boldsymbol{\Delta}, \beta \rangle$ where:*

- *R_0 is the initial goal recognition model,*
- *$\boldsymbol{\Delta}$ are the possible sensor extensions, and*
- *β is a budget on the number of allowed extensions.*

We want to find a set $\Delta \subseteq \boldsymbol{\Delta}$ of up to β sensor extensions to apply to R_0 offline to minimize the *wcd*. This objective is formally defined below, where $wcd^{min}(T)$ is the minimum *wcd* achievable in a GRD-APK model T, and R^{Δ} is the goal recognition model that results from applying set Δ to R.

$$\begin{aligned} wcd^{min}(T) = \min_{\Delta \subseteq \boldsymbol{\Delta}} \quad & wcd(R_0^{\Delta}) \\ s.t. |\Delta| \leq \beta \end{aligned} \tag{1}$$

Any solution to Equation 1 is *optimal*, i.e., it achieves the minimal *wcd* possible. It is *strongly optimal* if it has minimum size among all optimal solutions, i.e., it includes the minimal number of extensions needed to minimize *wcd*.

The *k-planner* and $K_{prudent}(P)$ Translation

A variety of solvers have been developed to solve a PPO-det problem (e.g., (Bonet and Geffner 2011; Muise, Belle, and McIlraith 2014; Brafman and Shani 2012b)), all of which can be used to represent the actor (and its set of possible policies) described in Definition 2. Specifically, Bonet and Geffner (2011) suggest the *k-planner* that follows the *planning under optimism* approach; the actor plans while making the most convenient assumptions about the values of (i.e., assigns a value to) hidden variables, executes the plan that is obtained from the resulting classical planning problem, and revises the assumptions and re-plans, if during the execution, an observation refutes the assumptions made.

To transform the PPO-det problem into a classical planning problem, the k-planner uses the $K(P)$ translation. At the core of the translation is the substitution of each literal L in the original problem with a pair of fluents KL and $K\neg L$, representing whether L is known to be true or false, respectively (Albore, Palacios, and Geffner 2009). Each original action $a \in \mathcal{A}$ is transformed into an equivalent action that replaces the use of every literal L ($\neg L$), with its corresponding fluent KL ($K\neg L$). Each observation (C, L) is translated into two deterministic sensing actions, one for each possible value of L. These sensing actions allow the solver to compute a plan while choosing preferred values of (making assumptions about) the unknown variables. For example, the actor can assume that a cell on its planned path has no pit (e.g., $K\neg PitAt(4, 1) = True$). Each invariant clause is translated into a set of actions, which we call *ramification actions*. These actions can be used to set the truth value of some variable, as new sensing information is collected from the environment. For example, a ramification action can be activated to deduce that a cell is safe when no breeze or stench is sensed in an adjacent cell.

The action set in the transformed problem is therefore $\mathcal{A}' = \mathcal{A}^{'}_{exe} \cup \mathcal{A}^{'}_{sen} \cup \mathcal{A}^{'}_{ram}$, where $\mathcal{A}^{'}_{exe}$ represents the transformed original set of actions, $\mathcal{A}^{'}_{sen}$ are the sensing actions and $\mathcal{A}^{'}_{ram}$ are the ramification actions. This representation captures the underlying planning problem at the knowledge level, accounting for the exploratory behavior of a partially informed agent.

Bonet and Geffner (2011) show that this linear translation of a PPO-det problem into a classical planning problem is sound and complete for *simple* PPO-det models with a connected state space. A PPO-det model is simple if the non-unary clauses in I are all invariant, and no hidden fluent appears in the body of a conditional effect. In connected state spaces every state is reachable from any other. In simple problems there is no information loss and the model is *monotonic*, i.e., for every fluent $f \in \mathcal{F}$, if f is known in a belief state b and $b^{'}$ is a belief reachable from b, then f is known in $b^{'}$. As a consequence, for every policy π and history h of length n it follows that the number of states in beliefs b_i is a monotonically decreasing function, i.e., $|b_i| \geq |b_{i+1}|$ for every $0 \leq i < n$.

A key issue to note about the $K(P)$ compilation is that all its actions, including sensing and ramification actions, have equal cost. This means that a cost-minimizing solution to the resulting classical planning problem may be one that favors increasing the cost to goal over the use of multiple ramification actions. As described in Example 1, we want a solver that can make optimistic assumptions, but chooses a minimal cost plan that requires making as few assumptions as possible. In addition, ramifications are not to be considered when calculating the cost to goal. We therefore suggest the $K_{prudent}(P)$ translation, which extends the uniform cost $K(P)$ translation by associating a cost function to each action in $\mathcal{A}'$. Specifically, every transformed action $a \in \mathcal{A}^{'}_{exe}$ is assigned a cost of 1, every sensing action (assumption) $a \in \mathcal{A}^{'}_{sen}$ is assigned a small cost of ϵ, and every ramification action $a \in \mathcal{A}^{'}_{ram}$ has 0 cost. When ϵ is small enough such that the accumulated cost of assumptions of any generated plan is guaranteed to be smaller than minimal diversion from an optimal plan, the cost-minimal plan achieved using this formulation complies with our requirements.

Methods for Information Shaping

In our search for an optimal design solution, we consider a sensor extension as *useful* with regards to a goal recognition model if it reduces *wcd*. Given a goal recognition model R and a sensor extension δ, we let R^{δ} denote the model that results from applying δ to the actor's sensor model $\mathcal{O}$, and define useful sensor extensions as follows.

Definition 8 *A modification δ is* **useful** *with regards to goal recognition model R if* $\text{wcd}(R^{\delta}) < \text{wcd}(R)$.

The challenge in information shaping comes from two sources. First, the number of possible information shaping options may be large, and evaluating the effect of each change may be costly, making it important to develop efficient search techniques. Second, the problem is non-monotonic, in that sensor extensions are not always useful, and providing more information may actually make recognition more difficult by increasing *wcd* (Example 1).

To address these challenges, we follow Keren, Gal, and Karpas (2018) and formulate the design process as a search in the space of modification sets $\Delta \subseteq \mathbf{\Delta}$. With a slight abuse of notation, we let R^{Δ} denote the model that results from applying the set Δ of sensor extensions to the actor's sensor model. The root node is the initial goal recognition model R_0 (and empty modification set), and the operators (edges) are the sensor extensions $\delta \in \mathbf{\Delta}$ that transition between models. Each node (modifications set Δ) is evaluated by $wcd(R_0^{\Delta})$, the *wcd* value of its corresponding model.

To calculate the *wcd* value of a model we need to find the maximal non-distinctive history. Recall that we assume the actor's solver is known to the recognizer, who can observe the actor's transition between states. We can therefore find the *wcd* value of a GR-APK model by first using the actor's solver to compute the policies to each of the goals. Then, starting at the initial state, we iteratively explore the non-distinctive policy prefixes, until its most distant boundary is found, and return its length (cost).

Design with CG-Pruning

The baseline approach for searching in modification space is *breadth first search* (BFS), using *wcd* to evaluate each node. Under the budget constraints, BFS explores modification sets of increasing size, using a closed-list to avoid the computation of pre-computed sets. The search halts if a model with $wcd = 0$ is found or if there are no more nodes to explore, and returns the shortest path (smallest modification set) to a node that achieves minimal *wcd*. This iterative approach is guaranteed to find a strongly optimal solution, i.e., a minimal set of modifications that minimizes *wcd*. However, it does not scale to larger problems.

To increase efficiency, pruning can be applied to reduce the size of the search space. Specifically, pruning is *safe* if at least one optimal solution remains unpruned (Wehrle and Helmert 2014). Keren, Gal, and Karpas (2018) offer a pruning technique for GRD settings where the actor is fully informed and guarantee it is safe if modifications cannot increase *wcd*. Since this condition does not hold in our setting, where sensor extensions can both increase and reduce *wcd*, we suggest a new pruning approach that eliminates useless modifications, and specify conditions under which it is safe.

The high level idea of our pruning technique is to transform the partially observable planning problem for each goal into its corresponding fully observable planning problem, and use off-the-shelf tools developed for fully observable planning in order to automatically detect information shaping modifications that are guaranteed not to have an effect on the actor's behavior.

Specifically, given a goal recognition model R, for every goal in $\mathcal{G}$, we use the $K(P)$ transformation (or its variant $K_{prudent}(P)$ introduced above) to transform the partially observable planning problem into a fully observable problem. We then construct the *causal graph* (Williams and Nayak 1997; Helmert 2006) of each transformed problem. According to Helmert (2006), the causal graph of a planning problem is a directed graph (V, E) where the nodes

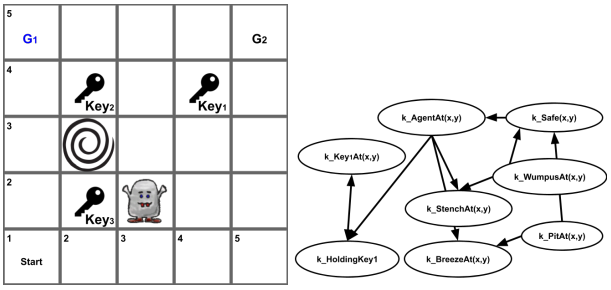


Figure 2: The Wumpus domain with keys

V represent the state variables and the edges E represent dependencies between variables, such that the graph contains a directed edge (v, v') for $v, v' \in V$ if changes in the value of v' can depend on the value of v. Specifically, to capture only the variables that are relevant to achieving the goal, the causal graph only contains ancestors of all variables that appear in the goal description. In our context, the variable set of the causal graph can either be the set of fluents of the transformed PPO-det problem, or the multi-valued variables extracted using *invariant synthesis*, which automatically finds sets of fluents among which exactly one is true at each state, and which can be assumed to represent the different values of a multi-valued variable. In any case, the casual graph $CG(G)$ of each goal $G \in \mathcal{G}$ captures all variables relevant for achieving the goal and the hierarchical dependencies between them. Recall that each sensor extension is characterized by an observation $o = (C, L)$ that is added to the actor's sensor model. Our pruning technique, dubbed *CG-Pruning*, prunes all sensor extensions for which the fluents corresponding to knowledge about L in the transformed problem (i.e. KL and $K\neg L$) do not occur in any of the casual graphs.

Example 2 *Consider Figure 2(left), depicting a modified version of Example 1, where the actor needs to collect a key to be able to access its goal (e.g., $PickedKey_1$ is needed to reach G_1). There are multiple keys distributed in the grid (e.g., $Key_1At(4,4)$), each needed for accessing a particular location. The actor initially knows a set of possible key locations for each key. When in a cell with a key, it senses it and can pick it up and use it to achieve its goal. In this scenario, the recognizer, with perfect information, can notify the actor about safe locations, as before, but also about the absence or presence of a particular key in some location. Applying the $K(P)$ transformation here creates fluents $KKey_iAt(x,y)$ for each key and location, representing whether the actor knows key i is at location (x,y), which is a precondition to picking up the key. Figure 2(right), show a part of the causal graph for G_1 that only includes variables concerning the location of its relevant key. By generating the causal graph to all goals, we automatically detect and prune sensor extensions regarding variables that do not appear in any of the causal graphs (e.g., the sensor extension that reveals the location of Key_3).*

In the following, we show that CG-Pruning is safe for GRD-APK settings where the actor uses the *k-planner* with an optimal planner to computes its plans. Since the actor uses the *k-planner*, it iteratively computes a policy at the initial state and every time an assumption made at a previous iteration is refuted. At each iteration, the current partially observable problem is transformed into its corresponding fully observable problem, and a new plan is computed and executed. This continues until the actor reaches a goal belief or a belief state with no applicable actions. For each model R and execution iteration i, we let $CG_i^R(G)$ represent the causal graph at iteration i and start our proof by showing that the causal graph at each iteration subsumes any causal graph of subsequent iterations.

Lemma 1 *For any model R and goal $G \in \mathcal{G}$, $CG_j^R(G)$ is a subgraph of $CG_i^R(G)$ for any i, j s.t. $0 \leq i < j$.*

Proof Sketch: The causal graph of iteration i captures all variables that appear in actions that may be applied in order to achieve a goal belief from the initial belief state at iteration i. This graph includes all the actions (and their corresponding variables) that may be applied from the belief reached at iteration j. ■

Lemma 1 guarantees that a variable that does not occur in $CG_0^R(G)$ for any goal $G \in \mathcal{G}$ will not occur in causal graphs of future iterations.

Next, we observe that when an optimal solver is used, a sensor extension that does not correspond to a variable in the initial causal graph of any goal is not useful.

Lemma 2 *For any model R and sensor extension δ that adds observation $o = (C, L)$ to $\mathcal{O}^{ac}$, if for all $G \in \mathcal{G}$, KL and $K\neg L$ are not in $CG_0^R(G)$, then δ is not useful w.r.t R.*

Proof Sketch: Bonet and Geffner (2011) show that the $K(P)$ transformation is sound and complete for simple problems with a connected space, which are the only problems we consider here. Helmert (2006) shows that any optimal plan can be acquired by ignoring variables that are not in the causal graph. Therefore, by pruning sensor extensions that are related to variables not on the causal graph, we are removing from the actor's planning graph sensing actions that would anyway not appear in any optimal plan (i.e., assumptions the actor would not make). Therefore, the behavior of an actor to any goal is not affected by such sensor extensions. Moreover,as the actor progresses and re-plans, no sensing action can be added to the actor's model. Consequently, the behavior w.r.t to any goal will not change, *wcd* will not change, and therefore δ is not useful w.r.t R. ■

Finally, we are ready to show that CG-Pruning is safe.

Theorem 1 *For any GRD-APK model $T = \langle R_0, \mathbf{\Delta}, \beta \rangle$, CG-Pruning is safe for an actor that uses the* k-planner *with an optimal planner.*

Proof Sketch: Lemma 2 guarantees, that under the assumptions we make, any sensor extension that adds observation

$o = (C, L)$ to $\mathcal{O}^{ac}$ and for which neither KL or $K\neg L$ appear in $CG_0^R(G)$, are not useful to any model reachable from R_0 via design and will not be part of a strongly optimal solution. Therefore CG-Pruning is safe.

■

Empirical Evaluation

Our objective is to evaluate both the effect sensor extensions have on *wcd* as well as the efficiency of CG-Pruning.We start by describing our dataset and empirical setup, and then discuss our initial results.

Dataset. We used five domains adapted from Bonet and Geffner (2011) and Albore, Palacios, and Geffner (2009).

- WUMPUS: corresponding to the setting in Example 1.
- WUMPUS-KEY: corresponding to Example 2.
- C-BALLS (Colored-balls): the actor navigates a grid to deliver balls of different and initially unknown colors to their per-color destinations.
- TRAIL: an agent must follow a trail to reach a destination, while sensing only the reachable cells surrounding it.
- Logistics: Packages are transported to their destinations, relying on sensing to reveal the packages in a location.

The adaptation from contingent planning to GRD-APK involves specifying for each instance the set of possible goals and sensor extensions (see Table 1 for details).

To support the design process, we use STRIPS (Fikes and Nilsson 1972) to specify the available modifications (and their effect). Sensor extensions are implemented as design actions that add to the initial state fluents that represent the true value of a variable.

Setup. We use the *k-replanner* (Bonet and Geffner 2011) as the actor's solver, with two variations. For the first, the $K(P)$ compilation was used together with the satisfying FF classical planner (FF) (Hoffmann and Nebel 2001). The second used the $K_{prudent}(P)$ compilation together with the optimal Fast-Downward (Helmert 2006) classical planner (FD), using the lm-cut heuristic (Helmert and Domshlak 2009).

In our computation of *wcd*, we also consider the prefixes of failed executions, since they represent valid agent behavior. The design process is implemented as a breadth-first search (BFS) in the space of modification sets, tested with and without CG-Pruning.

We use 30 instances for each domain, and a design budget of $1-2$. Each execution had a time limit of 20 minutes and is capped at 1000 search steps (each corresponding to a design set), whichever was first.

To parse the design file, we adopt the parser of *pyperplan* (Alkhazraji et al. 2016), which provide for each modification set (representing a GRD-APK model and a node in our search) the set of successors (applicable modifications) and the model that results from applying each modification.

Results. Tables 2 and 3 summarize the results for both approaches (No Pruning vs. CG-Pruning) for the FD and FF solvers, respectively. For each domain and design budget ($b = 1$ and $b = 2$), the tables shows 'sol' as the fraction of instances completed within the time and resource bounds. For instances completed by both approaches 'Δ-*wcd*' is the average *wcd* reduction achieved via design, i.e., the *wcd* difference between the original setting and one where sensor extensions are applied (note that since CG-Pruning is safe 'Δ-*wcd*' is the same for both approaches). In parenthesis we show 'Δ-*wcd*' over all instances, including those that timed out. The average calculation time (in seconds) for each approach is indicated by 'time', and 'nodes' is the average number of nodes evaluated on all instances. 'NA' represents settings for which no instance completed. In Table 2 we excluded C-BALLS and TRAIL, since no problem completed for both domains.

Our results show that design via information shaping reduces *wcd* for all domains, with a reduction of 9.12 (about half) for C-BALLS. By excluding futile sensor extensions, for all domains CG-Pruning reduces the number of nodes explored and computation time for completed problems. For WUMPUS, WUMPUS-KEY and LOGISTICS using FF, CG-Pruning also increases the ratio of solved problems.

The results show the potential of our pruning approach. However, many instances were not completed for FD, failing in some cases to complete the solution of the initial setting. To achieve more results for the optimal case, and hopefully a stronger indication of the benefit of our approach in such settings, we intend to add additional domains to our dataset and explore different heuristics used to guide the optimal search. We also intend to enhance pruning further. Specifically, using the plan the actor intends to execute with regards to each goal, we can prune sensor extensions that correspond to as-

	Possible Goals	Sensor Extensions
WUMPUS	gold locations	safe cells
WUMPUS-KEY	gold locations	safe cells or locations with / without keys
C-BALLS	ball distribution	locations without a ball
TRAIL	final stone locations	locations with / without stones
LOGISTICS	package destination	package locations

Table 1: Possible goals and design options for each domain.

		No Pruning				CG-Pruning			
	budget	sol	Δ*wcd*	time	nodes	sol	Δ*wcd*	time	nodes
WUMPUS	b=1	0.1	0.0 (1.8)	92.84	14.0	0.1	0.0 (1.8)	76.96	11.0
	b=2	0.1	0.0 (1.8)	663.71	106.0	0.1	0.0 (1.8)	421.27	67.0
WUMPUS-KEY	b=1	1.0	0.2(0.57)	16.05	4.1	1.0	0.2(0.57)	12.59	3.3
	b=2	0.71	0.2(0.57)	238.74	39.5	0.72	0.2(0.57)	200.34	28.8
LOGISTICS	b=1	0.14	8.0 (11.83)	1330.33	3.0	0.14	8.0(11.83)	1042.52	2.0
	b=2	NA	NA	NA	NA	NA	NA	NA	NA

Table 2: Results per domain for (optimal) the FD solver

		No Pruning				CG-Pruning			
	budget	sol	Δ*wcd*	time	nodes	sol	Δ*wcd*	time	nodes
WUMPUS	b=1	1.0	0.0 (3.0)	88.02	16.0	1.0	0.0 (6.0)	59.98	11.0
	b=2	0.25	0.0 (3.0)	697.31	137.0	1.0	0.0 (6.0)	351.37	67.0
WUMPUS-KEY	b=1	1.0	4.33 (4.33)	16.71	13.55	1.0	4.33 (4.33)	13.35	10.56
	b=2	0.8	3.95 (3.95)	85.73	54.56	1.0	3.95 (3.95)	75.55	42.55
C-BALLS	b=1	0.8	9.12 (9.2)	36.61	37.03	0.8	9.12 (9.2)	38.75	37.03
	b=2	0.8	11.5 (10.83)	30.19	22.01	0.8	11.5 (10.83)	30.19	22.01
TRAIL	b=1	1.0	0.0 (0.0)	14.71	28.0	1.0	0.0 (0.0)	12.97	26.5
	b=2	1.0	0.0 (0.0)	195.39	407.0	1.0	0.0 (0.0)	173.21	365.5
LOGISTICS	b=1	0.42	3.01 (4.14)	22.90	61.5	1.0	3.01 (4.14)	19.41	42.67
	b=2	0.28	9.05 (9.27)	133.68	234.4	0.86	9.05 (9.17)	112.89	175.1

Table 3: Results per domain for the (satisfying) FF solver

sumptions already made by the actor, and show that they will not reduce the *wcd*.

Related Work

Goal Recognition Design (GRD), a special case of *environment design* (Zhang, Chen, and Parkes 2009), was first introduced by Keren et al. (2014) to account for optimal fully observable agents in deterministic domains. This work was later extended to a variety of GRD settings, including accounts for sub-optimal actors (Keren, Gal, and Karpas 2015), stochastic environments (Wayllace et al. 2016), adversarial actors that try to conceal their goal (Ang et al. 2017), and a partially informed recognizer (Keren, Gal, and Karpas 2016a; 2016b; 2018). In the latter case, sensor refinement is applied to enhance the recognizer's sensor model.

Common to all previous GRD work is the assumption that actors have perfect observability of their environment. Our work is the first to generalize GRD to account for a partially informed actor and to suggest new information shaping modifications, implemented as sensor extensions applied to the actor's sensor model, as a way to reduce *wcd*.

Efficient communication via selective information revelation is fundamental to various multi agent settings, e.g., (Xuan, Lesser, and Zilberstein 2001; Wu, Zilberstein, and Chen 2011; Unhelkar and Shah 2016; Dughmi and Xu 2016). This work is the first to use information shaping as a one time and offline intervention that is performed in order to facilitate goal recognition.

Conclusion

We introduced GRD for a partially informed actor and a perfectly informed recognizer, who can share information about the domain with the actor. We formalized the information shaping problem as one of minimizing worst-case distinctiveness, and presented new sensor extension modifications, used to enhance recognition. We studied the use of breadth first search to search the space of applicable sensor extensions, developing a safe pruning approach to improve efficiency. To the best of our knowledge, this is the first paper to suggest using techniques developed for classical planning toward the design of algorithms for goal recognition of partially informed planning agents. Our results on a set of standard benchmarks show that *wcd* can be reduced via information shaping and demonstrate the efficiency of our approach.

There are many ways to extend this work. First, we use qualitative contingent planning models to represent the partially informed agent and its belief states. It would be interesting to extend this work to use Partially Observable Markov Descision Process (POMDP) models (Kaelbling, Littman, and Cassandra 1998) to represent the actor. Another interesting direction is to consider settings where the actor is aware of the recognizer's presence. Specifically, our approach can be adopted to "transparent planning" (MacNally et al. 2018), where actors choose behaviors that facilitate recognition. These models rely on partially informed agents to be able to choose a behavior that maximizes the information conveyed about their intentions. In such settings, GRD can be viewed as a complementary approach, that can be applied to alleviate the need to completely rely on the actor, and reduce the number of non-distinctive behaviors. Another variation worth exploring is an interactive setting, where the recognizer can decide which information to reveal based on the actor's actual progress. This would be especially relevant to many realistic settings where the recognizer cannot be assumed to have perfect information about the solver used by the actor. Finally, while we focus on pruning as a way to increase efficiency, other options are possible. In particular, heuristics can be used to estimate the value of a modification, and lead the search in promising directions.

Acknowledgements

The authors thank Miquel Ramirez, Nir Lipovetzky and Blai Bonet for their helpful comments and suggestions.

References

Albore, A.; Palacios, H.; and Geffner, H. 2009. A translation-based approach to contingent planning. In *IJCAI*.

Alkhazraji, Y.; Frorath, M.; Grutzner, M.; Liebetraut, T.; Ortlieb, M.; Seipp, J.; Springenberg, T.; Stahl, P.; Wulfing, J.; Helmert, M.; and Mattmuller, R. 2016. Pyperplan: https://bitbucket.org/malte/pyperplan.

Ang, S.; Chan, H.; Jiang, A. X.; and Yeoh, W. 2017. Game-theoretic goal recognition models with applications to security domains. In *International Conference on Decision and Game Theory for Security*.

Boddy, M. S.; Gohde, J.; Haigh, T.; and Harp, S. A. 2005. Course of action generation for cyber security using classical planning. In *ICAPS*.

Bonet, B., and Geffner, H. 2011. Planning under partial observability by classical replanning: Theory and experiments. In *IJCAI*.

Brafman, R., and Shani, G. 2012a. A multi-path compilation approach to contingent planning. In *AAAI*.

Brafman, R., and Shani, G. 2012b. Replanning in domains with partial information and sensing actions. *Journal of Artificial Intelligence Research (JAIR)* 45.

Carberry, S. 2001. Techniques for plan recognition. *User Modeling and User-Adapted Interaction* 11.

Cohen, P. R.; Perrault, C. R.; and Allen, J. F. 1981. Beyond question-answering. Technical report, DTIC Document.

Dughmi, S., and Xu, H. 2016. Algorithmic bayesian persuasion. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*.

Fikes, R. E., and Nilsson, N. J. 1972. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence 1972*.

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

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

Hoffmann, J., and Nebel, B. 2001. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 14.

Kaelbling, L. P.; Littman, M. L.; and Cassandra, A. R. 1998. Planning and acting in partially observable stochastic domains. *Artificial intelligence* 101(1-2).

Kautz, H., and Allen, J. F. 1986. Generalized plan recognition. In *AAAI*, volume 86.

Kautz, H.; Etzioni, O.; Fox, D.; Weld, D.; and Shastri, L. 2003. Foundations of assisted cognition systems. Technical report, University of Washington.

Keren, S.; Gal, A.; and Karpas, E. 2014. Goal recognition design. In *ICAPS*.

Keren, S.; Gal, A.; and Karpas, E. 2015. Goal recognition design for non optimal agents. In *AAAI*.

Keren, S.; Gal, A.; and Karpas, E. 2016a. Goal recognition design with non-observable actions. In *AAAI*.

Keren, S.; Gal, A.; and Karpas, E. 2016b. Privacy preserving plans in partially observable environments. In *IJCAI*.

Keren, S.; Gal, A.; and Karpas, E. 2018. Strong stubborn sets for efficient goal recognition design. In *ICAPS*.

Levine, S. J., and Williams, B. C. 2014. Concurrent plan recognition and execution for human-robot teams. In *ICAPS*.

MacNally, A.; Lipovetzky, N.; Ramirez, M.; and Pearce, A. 2018. Action selection for transparent planning. In *Proceedings of the Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*.

Muise, C. J.; Belle, V.; and McIlraith, S. A. 2014. Computing contingent plans via fully observable non-deterministic planning. In *AAAI*.

Pereira, R. F.; Oren, N.; and Meneguzzi, F. 2017. Landmark-based heuristics for goal recognition. In *AAAI*.

Ramirez, M., and Geffner, H. 2010. Probabilistic plan recognition using off-the-shelf classical planners. In *AAAI*.

Russell, S. J., and Norvig, P. 2016. *Artificial intelligence: a modern approach*. Pearson Education.

Sukthankar, G.; Geib, C.; Bui, H. H.; Pynadath, D.; and Goldman, R. P. 2014. *Plan, activity, and Intent Recognition: Theory and practice*. Newnes.

Unhelkar, V. V., and Shah, J. A. 2016. Contact: Deciding to communicate during time-critical collaborative tasks in unknown, deterministic domains. In *AAAI*.

Wayllace, C.; Hou, P.; Yeoh, W.; and Son, T. C. 2016. Goal recognition design with stochastic agent action outcomes". In *IJCAI*.

Wehrle, M., and Helmert, M. 2014. Efficient stubborn sets: Generalized algorithms and selection strategies. In *ICAPS*.

Williams, B. C., and Nayak, P. P. 1997. A reactive planner for a model-based executive. In *IJCAI*, volume 97.

Wu, F.; Zilberstein, S.; and Chen, X. 2011. Online planning for multi-agent systems with bounded communication. *Artificial Intelligence* 175(2).

Xuan, P.; Lesser, V.; and Zilberstein, S. 2001. Communication decisions in multi-agent cooperation: Model and experiments. In *Proceedings of the fifth international conference on Autonomous agents*.

Zhang, H.; Chen, Y.; and Parkes, D. C. 2009. A general approach to environment design with one agent. In *IJCAI*.

Width-Based Lookaheads Augmented with Base Policies for Stochastic Shortest Paths

Stefan O'Toole, Miquel Ramirez, Nir Lipovetzky, Adrian Pearce
The University of Melbourne, Melbourne, Australia
stefan@student.unimelb.edu.au, {miquel.ramirez, nir.lipovetzky, adrianrp}@unimelb.edu.au

Abstract

Sequential decision problems for real-world applications often need to be solved in real-time, requiring algorithms to perform well with a restricted computational budget. *Width-based* lookaheads have shown state-of-the-art performance in classical planning problems as well as over the Atari games with tight budgets. In this work we investigate width-based lookaheads over Stochastic Shortest paths (SSP). We analyse why width-based algorithms perform poorly over SSP problems, and overcome these pitfalls proposing a method to estimate costs-to-go. We formalize width-based lookaheads as an instance of the *rollout* algorithm, give a definition of width for SSP problems and explain its sample complexity. Our experimental results over a variety of SSP benchmarks show the algorithm to outperform other state-of-the-art *rollout* algorithms such as UCT and RTDP.

Keywords: width-based planning, finite-horizon MDPs, rollout algorithm, base policies

Introduction

Model-based lookahead algorithms provide the ability to autonomously solve a large variety of sequential decision making problems. Lookaheads search for solutions by considering sequences of actions that can be made from the current state up to a certain time into the future. For real-world applications decisions often need to be computed in real-time, requiring algorithms to perform with a restricted computational budget. Limiting search in this way can result in considering states and trajectories which do not provide useful information. To address this, lookaheads can be augmented with heuristics that estimate costs-to-go to prioritise states and trajectories, and have been shown to perform well where computation budgets are restricted (Eyerich, Keller, and Helmert 2010).

This paper is concerned with Stochastic Shortest Path (SSP) problems which are often used to compare and evaluate search algorithms. We consider the *width-based* family of planning algorithms, first introduced by Lipovetzky and Geffner (2012), which aim to prioritise the exploration of *novel* areas of the state space. Two width-based planners, Lipovetzky and Geffner's breadth-first search, IW(1), and the depth-first search, Rollout-IW(1) (Bandres, Bonet, and Geffner 2018), are investigated on SSP problems. We first provide the necessary background for SSP problems and width-based algorithms, while also formalising width-based algorithms as instances of the *rollout* algorithm (Bertsekas 2017). We then show the motive to augment width-based lookaheads with cost estimates on SSP problems, define the width of SSP problems and propose a novel width-based algorithm that estimates costs-to-go by simulating a general base policy. Our experimental study shows that the algorithm compares favourably to the original Rollout-IW(1) algorithm and to other state-of-the-art instances of the *rollout* algorithm.

Optimal Control and Dynamic Programming

We concern ourselves with the problem of decision under stochastic uncertainty over a finite number of stages, which we characterise following closely the presentation of Bertsekas (2017). We are given a discrete-time dynamic system

$$\mathbf{x}_{k+1} = f_k(\mathbf{x}_k, u_k, \mathbf{w}_k), \quad k = 0, 1, \ldots, N-1 \qquad (1)$$

where the *state* $\mathbf{x}_k$ is an element of a space $S_k \subset \mathbb{R}^d$, the control u_k is an element of space $C_k \subset \mathbb{N}$, and the random *disturbance* $\mathbf{w}_k$ is an element of a space $D_k \subset \mathbb{R}^m$ [1]. The *control* u_k is constrained to take values in a given non-empty subset $U(\mathbf{x}_k) \subset C_k$, which depends on the current state $\mathbf{x}_k$, so that $u_k \in U_k(\mathbf{x}_k)$ for all $\mathbf{x}_k \in S_k$ and k. The random disturbance $\mathbf{w}_k$ is characterised by a probability distribution $P_k(\cdot|\mathbf{x}_k, u_k)$ that may depend explicitly on $\mathbf{x}_k$ and u_k but not on the values of previous disturbances $\mathbf{w}_{k-1}, \ldots, \mathbf{w}_0$. We consider the class of *policies*, or *control laws*, corresponding to the sequence of functions

$$\pi = \{\mu_0, \ldots, \mu_{N+1}\} \qquad (2)$$

where μ_k maps states $\mathbf{x}_k$ into controls $u_k = \mu_k(\mathbf{x}_k)$ and is such that $\mu_k(\mathbf{x}_k) \in U(\mathbf{x}_k)$ for all $\mathbf{x}_k \in S_k$. Such policies will be called *admissible*. Given an initial state x_0 and admissible policy π, the states $\mathbf{x}_k$ and disturbances $\mathbf{w}_k$ are random variables with distributions defined through the system equation

$$\mathbf{x}_{k+1} = f_k(\mathbf{x}_k, \mu_k(\mathbf{x}_k), \mathbf{w}_k), \quad k = 0, 1, \ldots, N-1 \qquad (3)$$

[1] We define states and disturbance as elements of subsets of the reals to avoid too specific assumptions on the structure of S_k, U_k and D_k.

Thus, for given functions g_f (terminal cost) and g the expected cost of π starting at $\mathbf{x}_0$ is

$$J_\pi(\mathbf{x}_0) = E\Big\{ g_f(\mathbf{x}_N) + \sum_{k=0}^{N-1} g(\mathbf{x}_k, \mu_k(\mathbf{x}_k), \mathbf{w}_k) \Big\} \quad (4)$$

where the expectation is taken over the random variables $\mathbf{w}_k$ and $\mathbf{x}_k$. An *optimal policy* π^* is one that minimises this cost

$$J_{\pi^*}(\mathbf{x}_0) = \min_{\pi \in \Pi} J_\pi(\mathbf{x}_0) \quad (5)$$

where Π is the set of all admissible policies. The optimal cost $J^*(\mathbf{x}_0)$ depends on $\mathbf{x}_0$ and is equal to $J_{\pi^*}(\mathbf{x}_0)$. We will refer to J^* as the *optimal cost* or *optimal value* function that assigns to each initial state $\mathbf{x}_0$ the cost $J^*(\mathbf{x}_0)$.

Stochastic Shortest Path

We use Bertsekas' (2017) definition, that formulates Stochastic Shortest Path (SSP) problems as the class of optimal control problems where we try to minimize

$$J_\pi(\mathbf{x}_0) = \lim_{N \to \infty} E_{w_k} \Big\{ \sum_{k=0}^{N-1} \alpha^k g(\mathbf{x}_k, \mu_k(\mathbf{x}_k), \mathbf{w}_k) \Big\}$$

with α set to 1 and we assume there is a *cost-free termination state* $\mathbf{t}$ which ensures that $J_\pi(\mathbf{x}_0)$ is finite. Once the system reaches that state, it remains there at no further cost, that is, $f(\mathbf{t}, u, \mathbf{w}) = \mathbf{t}$ with probability 1 and $g(\mathbf{t}, u, \mathbf{w}) = 0$ for all $u \in U(\mathbf{t})$. We note that the optimal control problem defined at the beginning of this section is a special case where states are pairs $(\mathbf{x}_k, k)$ and all pairs $(\mathbf{x}_N, N)$ are lumped into termination state $\mathbf{t}$.

In order to guarantee termination with probability 1, we will assume that there exists an integer m such that there is a positive probability that $\mathbf{t}$ will be reached in m stages or less, regardless of what π is being used and the initial state $\mathbf{x}_0$. That is, for all admissible policies and $i = 1, ..., m$ it holds

$$\rho_\pi = \max_i P\{\mathbf{x}_m \neq \mathbf{t} \mid \mathbf{x}_0 = \mathbf{x}_i, \pi\} < 1 \quad (6)$$

A policy π will be *proper* if the condition above is satisfied for some m, and *improper* otherwise.

The Rollout Algorithm

A particularly effective on-line approach to obtain suboptimal controls is *rollout*, where the optimal cost-to-go from current state $\mathbf{x}_k$ is *approximated* by the cost of some suboptimal policy and a d-step lookahead strategy. The seminal RTDP (Barto, Bradtke, and Singh 1995) algorithm, is an instance of the rollout strategy where the lookahead is uniform, $d = 1$, and controls $\bar{\mu}(\mathbf{x}_k)$ selected at stage k and for state $\mathbf{x}_k$ are those that attain the minimum

$$\min_{u_k \in U(\mathbf{x}_k)} E\Big\{ g_k(\mathbf{x}_k, u_k, \mathbf{w}_k) + \tilde{J}_{k+1}(f_k(\mathbf{x}_k, u_k, \mathbf{w}_k)) \Big\} \quad (7)$$

where $\tilde{J}_{k+1}$ is an approximation on the optimal cost-to-go J^*_{k+1}. If the approximation is from below, we will refer to it as a *base heuristic*, and can either be problem specific (Eyerich, Keller, and Helmert 2010), domain independent (Bonet and Geffner 2003; Yoon, Fern, and Givan 2007) or *learnt* from interacting with a simulator (Mnih et al. 2015). Alternatively, $\tilde{J}_{k+1}$ can be defined as approximating the cost-to-go of a given suboptimal policy π, referred to as a *base policy*, where estimates are obtained via *simulation* (Rubinstein and Kroese 2017). We will denote the resulting estimate of cost-to-go as $H_k(\mathbf{x}_k)$[2]. The result of combining the lookahead strategy and the base policy or heuristic is the *rollout policy*, $\bar{\pi}$ $\{\bar{\mu}_0, \bar{\mu}_1, \ldots, \bar{\mu}_{N-1}\}$ with associated cost $\bar{J}(\mathbf{x}_k)$. Such policies have the property that for all $\mathbf{x}_k$ and k

$$\bar{J}_k(\mathbf{x}_k) \leq H_k(\mathbf{x}_k) \quad (8)$$

when H_k is approximating from above the cost-to-go of a policy, as shown by Bertsekas (2017) from the DP algorithm that defines the costs of both the base and the rollout policy. To compute at time k the rollout control $\bar{\mu}(\mathbf{x}_k)$, we compute and minimize over the values of the Q-factors of state and control pairs $(\mathbf{x}_l, u_l)$,

$$Q_l(\mathbf{x}_l, u_l) = E\big\{ g_l(\mathbf{x}_l, u_l, \mathbf{w}_l) + Q_{l+1}(f_l(\mathbf{x}_l, u_l, \mathbf{w}_l)) \big\} \quad (9)$$

for admissible controls $u_l \in U(\mathbf{x}_l)$, $l = k + i$, with $i = 0, ..., d - 1$, and

$$Q_l(\mathbf{x}_l) = E\big\{ H_l(\mathbf{x}_l) \big\} \quad (10)$$

for $l = k + d$. In this paper we make a number of assumptions to ensure the viability of lookaheads with $d > 1$. We will assume that we can *simulate* the system in Eq. 3 under the base policy, so we can generate sample system trajectories and corresponding costs consistent with probabilistic data of the problem. We further assume that we can reset the simulator of the system to an arbitrary state. Performing the simulation and calculating the rollout control still needs to be possible within the real-time constraints of the application, which is challenging as the number of Q-factors to estimate and minimizations to perform in Equations 9-10 is exponential on the average number of controls available per stage and d, the maximum depth of the lookahead. We avoid the blowup of the size of the lookahead by cutting the recursion in Equation 9 and replacing the right hand side by that of Equation 10. As detailed in the next section, we will do this when reaching states $\mathbf{x}_l$ that are deemed not to be *novel* according to the notion of structural *width* by Lipovetzky and Geffner (2012). This results in a selective strategy alternative to the upper confidence bounds (Auer, Cesa-Bianchi, and Fischer 2002) used in popular instances of Monte-Carlo Tree Search (MCTS) algorithms like Kocsis and Szepesvari's (2006) UCT, that also are instances of the rollout algorithm (Bertsekas 2017).

Width-Based Lookaheads

We instantiate the *rollout* algorithm with an l-step, depth-selective *lookahead* policy using *Width-based Search* (Lipovetzky and Geffner 2012). These algorithms both focus the lookahead and have good

[2]We use the subindex k to emphasize that the result of simulating a policy depends on the time step.

any-time behaviour. When it comes to prioritisation of expanding states, width-based methods select first states with novel valuations of features defined over the states (Lipovetzky, Ramirez, and Geffner 2015; Geffner and Geffner 2015). The most basic width-based search algorithm is $IW(1)$, a plain *breadth-first search*, guaranteed to run in *linear time and space* as it only expands *novel* states. A state $\mathbf{x}_l$ is *novel* if and only if it encounters a state variable [3] $x^i \subset \mathbb{R}$, whose value $v \in D(x^i)$, where $D(x^i)$ is the domain of variable x^i, has not been seen before in the current search. Note that novel states are independent of the objective function used, as the estimated cost-to-go J is not used to define the novelty of the states. IW(1) has recently been integrated as an instance of a *rollout* algorithm, and has been shown to perform well with respect to learning approaches with almost real-time computation budgets over the Atari games (Bandres, Bonet, and Geffner 2018).

Depth-First Width-Based Rollout

The breadth-first search strategy underlying IW(1) ensures a state variable x^i is seen for the first time through the shortest sequence of control steps, i.e. the shortest path assuming uniform costs $g(\mathbf{x}, u, \mathbf{w})$.[4] On the other hand, *depth-first* rollout algorithms cannot guarantee this property in general. Rollout IW (RIW) changes the underlying search of IW into a depth-first rollout. In order to ensure that RIW(1) considers a state to be novel *iff* it reaches at least one value of a state variable x^i_l through a shortest path, we need to adapt the definition of novelty. Intuitively, we need to define a set of state features to emulate the property of the breadth-first search strategy. Let $d(x^i, v)$ be the best upper bound known so far on the shortest path to reach each value $v \in D(x^i)$ of a state variable from the root state $\mathbf{x}_k$. Initially $d(x^i, v) = N$ for all state variables, where N is the horizon which is the maximum search depth allowed for the lookahead, thus denoting no knowledge initially. When a state $\mathbf{x}_l$ is generated, $d(x^i, v)$ is set to l for all state variables where $l < d(x^i, v)$.

Since RIW(1) always starts each new rollout from the current state $\mathbf{x}_k$, in order to prove a state $\mathbf{x}_l$ to be novel we have to distinguish between $\mathbf{x}_l$ being already in the lookahead tree and $\mathbf{x}_l$ being new. If $\mathbf{x}_l$ is *new* in the tree, to conclude it is novel, it is sufficient to show that there exists a state variable x^i whose known shortest path value $d(x^i, v) > l$. If $\mathbf{x}_l$ is *already* in the tree, we have to prove the state contains at least one state variable value x^i whose shortest path is $l = d(x^i, v)$, i.e. state $\mathbf{x}_l$ is still novel and on the shortest path to x^i. Otherwise the rollout is terminated.

In order to ensure the termination of RIW(1), non-novel states are marked with a *solved* label. The label is back-propagated from a state $\mathbf{x}_{l+1}$ to $\mathbf{x}_l$ if all the admissible control inputs $u \in U(\mathbf{x}_l)$ yield states $\mathbf{x}_{l+1} = f_l(\mathbf{x}_l, u, \mathbf{w}_l)$ already labeled as *solved*. RIW(1) terminates once the root state is labeled as *solved* (Bandres, Bonet, and Geffner 2018). Non-novel states $\mathbf{x}_l$ are treated as terminals and their cost-to-go is set to 0. This can induce a bias towards non-novel states rather than true terminal states. In the next section we investigate how to overcome the ill-behaviour of RIW(1) when a state x_l is non-novel. We discuss the importance of estimating upper-bounds on the cost-to-go $H_l(\mathbf{x}_l)$ instead of assigning termination costs. This turns out to be essential for RIW(1) over SSPs.

[3] In order to use the notion of novelty, we assume state spaces S to be stationary.

[4] This can easily be generalized to non-uniform costs by using Dijkstra's algorithm instead.

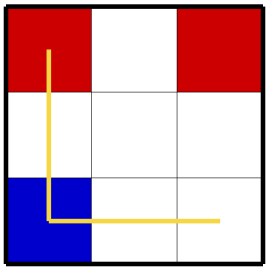


Figure 1: 3x3 `GridWorld` problem in which the blue square is the agent's initial position and the red squares show two goal locations. The yellow lines represent two action trajectories the agent can perform from the initial state.

Width-Based Lookaheads on SSPs

Despite the successes of *width-based* algorithms on a variety of domains including the Atari-2600 games (Lipovetzky, Ramirez, and Geffner 2015; Bandres, Bonet, and Geffner 2018), the algorithms, as will be shown, have poor performance on SSP problems. We illustrate this with two scenarios. First, width-based lookaheads prefer trajectories leading to non-novel states over longer ones that reach a goal. Second, and specific to depth-first width-based lookaheads, we show that useful information is ignored. We can demonstrate these scenarios using a simple SSP problem with uniform and unitary action costs, shown in Figure 1. The task is to navigate to a goal location using the least number of left, right, up or down actions. Any action that would result in the agent moving outside of the grid produces no change in its position. The features used by the width-based planners are the coordinates for the current agent position. Both IW(1) and RIW(1) algorithms, given a sufficient budget, would result in the lookahead represented by yellow lines in Figure 1. As expected, both lookaheads contain the shortest paths to make each feature of the problem true. For both IW(1) and RIW(1), we back up the costs found in the lookahead starting from terminal and non-novel states. In this instance a move down or left from the agent's initial state has no effect, thus immediately producing a non-novel state.When backing up values, down and left have an expected cost of 1, which is less than the optimal cost of 2 for up, the action that leads to the top left goal state. This prevents both IW(1) and RIW(1) from ever achieving the goal, as they keep selecting those useless actions. Furthermore, if the goal is the top right location in Figure 1, RIW(1)'s random action selection can generate a trajectory that reaches the goal. Yet, trajectories leading to the goal are pruned away, as non-novel states in later considered trajectories are treated as terminals, again resulting in the lookahead represented by the yellow lines in Figure 1.

Novelty, Labeling and Width of SSPs

Bandres et al. (2018) introduced the algorithm RIW in the context of deterministic transition functions. In this section we discuss its properties in the context of SSPs.

The set of features used to evaluate the novelty of a state is $F = \{(v, i, d) \mid v \in D(\mathbf{x}^i)\}$ where $D(\mathbf{x}^i)$ is the domain of variable $\mathbf{x}^i$, and d is a possible shortest path distance. Note that the horizon N is the upper-bound of d. The maximum number of novel states is $O(|F|)$, as the maximum number of shortest paths for a feature $(v, i, \cdot) \in F$ is N. That is, in the worst case we can improve the shortest path for $(v, i, \cdot)$ by one control input at a time.

The labeling of nodes ensures the number of rollouts from the initial state in RIW(1) is at most $O(|F| \times b)$, where $b = max_{\mathbf{x}_l}|U(\mathbf{x}_l)|$ is the maximum number of applicable control variables in a state, i.e. maximum branching factor. When the labeling is applied to stochastic shortest path problems, the resulting lookahead tree is a relaxation of the original SSP, as it allows just one possible outcome of a control input. Alternatively, one can back-propagate the label *solved* to a state $\mathbf{x}_l$ iff 1) all admissible control inputs $u \in U(\mathbf{x}_l)$ have been applied resulting in states labeled as *solved*, and 2) the tree contains all the possible resulting states of each control input $u \in U(\mathbf{x}_l)$. We refer to this new strategy to back-propagate labels as λ-labeling. We denote as λ the maximum number of states that can result from applying $u \in U(\mathbf{x}_{l-1})$ in a state $\mathbf{x}_l$. That is, $\lambda = max_{\mathbf{x},u,w}|f(\mathbf{x}, u, \mathbf{w})|$. RIW(1) with λ-labeling will terminate after at most $O(|F| \times b \times \lambda)$ rollouts.

Furthermore, we can reconcile the notion of *width* over classical planning problems (Lipovetzky and Geffner 2012) with SSPs. A terminal state $\mathbf{t}$ made of features $f \in F$ has width 1 iff there is a trajectory $\mathbf{x_0}, u_0, \ldots, u_{n-1}, \mathbf{x}_n$ for $n \leq N$ where $\mathbf{x}_n = \mathbf{t}$, such that for each $\mathbf{x}_j$ in the trajectory 1) the prefix $\mathbf{x_0}, u_0, \ldots, u_{j-1}, \mathbf{x}_j$ reaches at least one feature $f_j = (v, i, d) \in F$ where all $(v, i, d') \in F$ for $d' < d$ are unreachable, i.e., it is a shortest path possible to reach a value in x^i, 2) any shortest path to f_j can be extended with a single control input u into a shortest path for a feature f_{j+1} complying with property 1) in state $\mathbf{x}_{j+1}$, and 3) the shortest path for f_n is also a shortest path for termination state $\mathbf{t}$. RIW(1) with the new labeling strategy is guaranteed to reach all width 1 terminal states $\mathbf{t}$.

Theorem 1. *Rollout IW(1) with λ-labeling is guaranteed to reach every width 1 terminal state $\mathbf{t}$ in polynomial time in the number of features F if $\lambda = 1$.*

If $\lambda = \infty$, RIW(1) will not propagate any *solved* label, and terminate only when the computational budget is exhausted.

For simplicity, we assumed shortest paths are equivalent to the shortest sequence of control inputs. To generalize to positive non-uniform costs, the distance d in the features should keep track of the cost of a path $\sum_i g(\mathbf{x}_i, u_i, \mathbf{w}_i)$ instead of its length, and the horizon be applied to the cost of the path.

Cost-to-go Approximation

The most successful methods for obtaining cost-to-go approximations have revolved around the idea of running a number of Monte Carlo simulations of a suboptimal base policy π (Ginsberg 1999; Coulom 2006). This amounts to generating a given number of samples for the expression minimized in Equation 7 starting from the states $\mathbf{x}_l$ over the set of admissible controls $u_l \in U(\mathbf{x}_l)$ in Equation 10, averaging the observed costs. Three main drawbacks (Bertsekas 2017) follow from this strategy. First, the costs associated with the generated trajectories may be wildly overestimating $J^*(\mathbf{x}_l)$ yet such trajectories can be very rare events for the given policy. Second, some of the controls u_l may be clearly dominated by the rest, not warranting the same level of sampling effort. Third, initially promising controls may turn out to be quite bad later on. MCTS algorithms aim at combining lookaheads with stochastic simulations of policies and aim at trading off computational economy with a small risk of degrading performance. We add two new methods to the MCTS family, by combining the multi-step, width-based lookahead strategy discussed in the previous section with two simulation-based cost-to-go approximations subject to a *computational budget* that limits the number of states visited by both the lookahead and base policy simulation.

Width-based Lookaheads with Random Walks

The first method, which we call RIW-RW, uses as the base policy a *random walk*, a stochastic policy that assigns the same probability to each of the controls u admissible for state $\mathbf{x}$, and is generally regarded as the default choice when no domain knowledge is readily available. A rolling horizon H is set for the rollout algorithm that follows from combining the RIW(1) lookahead with the simulation of random walks. The maximal length of the latter is set to $H - l$, where l is the depth of the non-novel state. Both simulations and the unrolling of the lookahead are interrupted if the computational budget is exhausted. While this can result in trajectories sometimes falling short from a terminal, it keeps a lid on the possibility of obtaining extremely long trajectories that eat into the computational budget allowed and preclude from further extending the lookahead or sampling other potentially more useful leaf states $\mathbf{x}_l$.

Worst-Case Estimates of Rollout Costs

One of the most striking properties of rollout algorithms is the *cost improvement* property in Equation 8, suggesting that upper bounds on costs-to-go can be used effectively to approximate the optimal costs J^*. Inspired by this, the second width-based MCTS method we discuss leverages the sampling techniques known as *stochastic enumeration* (SE) (Rubinstein and Kroese 2017) to obtain an *unbiased estimator* for upper bounds on costs-to-go, or in other words, estimates the maximal costs a stochastic rollout algorithm with a large depth lookahead can incur.

SE methods are inspired by a classic algorithm by D. E. Knuth to estimate the maximum search effort by backtracking search (1975). Knuth's algorithm estimates the total cost of a tree T with root u keeping track of two quantities, C the estimate of the total cost, and D the expectation on the number of nodes in T at any given level of the tree, and the number of terminal nodes once the algorithm terminates. Starting with the root vertex u and $D \leftarrow 1$, the algorithm proceeds by updating D to be $D \leftarrow |\mathcal{S}(u)|D$ and choosing

randomly a vertex v from the set of successors $\mathcal{S}(u)$ of u. The estimate C is then updated $C \leftarrow C + c(u, v)D$ using the cost of the edge between vertices u and v. These steps are then iterated until a vertex v' is selected s.t. $\mathcal{S}(v') = 0$. We observe that Knuth's C quantity would correspond to the worst-case cost-to-go $\bar{J}(\mathbf{x})_k$ of a rollout algorithm using a lookahead strategy with d set to the rolling horizon H and the trivial base heuristic that assigns 0 to every leaf state. Furthermore, we assume that the algorithm either does not find any terminals within the limits imposed by the computational budget assigned, or if it finds one such state, it is too the very last one being visited. Lookaheads define trees over states connected by controls, edge costs $c(u, v)$ correspond directly with realisations of the random variable $g(\mathbf{x}, u, \mathbf{w})$ and the set of successors $\mathcal{S}(v)$ of a vertex corresponds with the set of admissible controls $U(\mathbf{x})$. While Knuth's algorithm estimates are an unbiased estimator, the variance of this estimator can be exponential on the horizon, as the accuracy of the estimator lies on the assumption that costs are evenly distributed throughout the tree (Rubinstein and Kroese 2017). In the experiments discussed next, we use Knuth's algorithm directly to provide $H_k(\mathbf{x}_k)$, adding the stopping conditions to enforce the computational budget and limiting the length of trajectories to $H - l$ as above. In comparison with simulating the random walk policy, the only overhead incurred is keeping up-to-date quantities C and D with two multiplications and an addition.

Experimental Study

Domains

To evaluate the different methods we use a number of `GridWorld` (Sutton and Barto 2018) domains, an instance of a SSP problem. The goal in `GridWorld` is to move from an initial position in a grid to a goal position. In each state 4 actions are available: to move up, down, left or right. Any action that causes a move outside of the grid results in no change to the agent's position. Actions have a cost of 1, with the exception of actions that result in reaching the goal state, that have a cost of 0. The complexity of `GridWorld` can be scaled through the size of the grid and the location and number of goals. `GridWorld` also allows for extensions, which we use to have domains with a stationary goal, moving goals, obstacles and partial observability. For each instance of the `GridWorld` domain we have a $d_0 \times d_1$ grid, and the state is the current location of the agent, $\mathbf{x} = (a_0, a_1)$ where a_i is the agent's position in dimension i. The transition function is formalised as

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{ef}_{u_k} \quad \text{if } \mathbf{x}_k + \mathbf{ef}_{u_k} \in S_{k+1} \wedge \mathbf{x}_k \notin T_k \tag{11}$$

where, $\mathbf{ef}$ specifies the change in the agent's position for each action, $T_k \subset S_k$ is the set of goal states and $\mathbf{x}_{k+1} = \mathbf{x}_k$ where the condition in Equation 11 is not met. The cost of a transition is defined as

$$g_k(\mathbf{x}_k, u_k, \mathbf{w_k}) = 0 \quad \text{if } \mathbf{x}_{k+1} \in T_{k+1} \tag{12}$$

otherwise, $g_k(\mathbf{x}_k, u_k, \mathbf{w_k}) = 1$.

For the **stationary goal** setting we have a single goal state which is positioned in the middle of the grid by dividing and rounding d_0 and d_1 by two. The problem setting with **moving goals**, has the set of goal states modified as follows

$$T_{k+1} = \{t_k + \delta_{t_k} \,|\, t_k \in T_k\} \quad \text{if } \mathbf{x}_k \notin T_k \tag{13}$$

where δ_{t_k} gives the relative change of the goal state t_k for the time step $k + 1$ and $T_{k+1} = T_k$ if $\mathbf{x}_k \in T_k$. We use $T_0 = \{(0, d_1 - 1), (d_0 - 1, 0)\}$, $d_0 = d_1$ and

$$\delta_{t_k} = \begin{cases} (1, -1) & \text{if } t_k = (0, d_1 - 1) \\ (-1, 1) & \text{if } t_k = (d_0 - 1, 0) \\ \delta_{t_{k-1}} & \text{otherwise} \end{cases} \tag{14}$$

Resulting in two goals starting at opposite corners of the grid moving back and forth on the same diagonal. The **obstacles** setting, uses the stationary goal, but modifies S_k such that,

$$S_k = \{(s_0,\, s_1) \,|\, 0 \leq s_0 < d_0,\, 0 \leq s_1 < d_1\} \setminus O \tag{15}$$

where $O \subset N^2$ and is the set of obstacles, that is grid cells in which the agent is not allowed.

Having partially observable obstacles in `GridWorld` provides an instance of the stochastic **Canadian Traveller Problem** (CTP) (Papadimitriou and Yannakakis 1991). The objective in CTP is to find the shortest path between one location in a road map to another, however, there is a known probability for each road in the map that due to weather conditions the road is blocked. A road in CTP can only be observed as being blocked or unblocked by visiting a location connected to it, and once a road status is observed the status remains unchanged. In terms of the `GridWorld` problem, each grid cell has a known probability of being a member of the obstacle set, O. The agent can only observe cells as being obstacles or not when it is in a neighbouring cell. Once a grid cell is observed it is then known that it is either an obstacle or not for the remaining duration of the episode.

John Langford designed two MDP problems[5] described as **Reinforcement Learning (RL) "Acid"** intended to be

Dim.	Alg.	Heu.	Simulator Budget 100	1000	10000
10	1Stp	Rnd.	29.6 ± 2.5	13.5 ± 1.6	7.5 ± 0.9
	UCT	Rnd.	29.0 ± 2.6	17.1 ± 2.0	13.3 ± 1.5
	RIW	NA	39.1 ± 2.8	38.1 ± 2.9	38.4 ± 2.9
		Rnd.	33.7 ± 2.5	$\mathbf{6.9 \pm 0.7}$	$\mathbf{4.7 \pm 0.4}$
20	1Stp	Rnd.	89.6 ± 3.7	59.8 ± 5.2	29.6 ± 3.1
	UCT	Rnd.	85.2 ± 4.3	72.7 ± 5.8	45.7 ± 4.4
	RIW	NA	79.8 ± 5.5	79.8 ± 5.5	80.2 ± 5.5
		Rnd.	88.2 ± 3.9	55.3 ± 5.2	$\mathbf{10.5 \pm 0.9}$
50	1Stp	Rnd.	215.2 ± 11.5	201.8 ± 13.5	177.9 ± 13.5
	UCT	Rnd.	220.4 ± 10.8	199.2 ± 13.5	190.6 ± 13.9
	RIW	NA	200.2 ± 13.8	200.2 ± 13.8	200.2 ± 13.8
		Rnd.	223.2 ± 10.4	199.9 ± 13.6	$\mathbf{145.5 \pm 12.9}$

Table 1: Average and 95% confidence interval for the cost on `GridWorld` with a *stationary* goal. Costs reported are from 200 episodes over 10 different initial states (20 episodes per initial state) of the `GridWorld` with a square grid with width and length equal to the dimension (Dim.) value. The horizon of each problem is 5 times the dimension value.

[5] https://github.com/JohnLangford/RL_acid

Dim.	Alg.	Heu.	Simulator Budget		
			100	1000	10000
10	1Stp	Rnd.	17.9 ± 2.1	10.1 ± 1.0	6.8 ± 0.5
	UCT	Rnd.	18.9 ± 2.2	11.0 ± 1.3	10.2 ± 1.1
	RIW	NA	39.8 ± 2.7	38.6 ± 2.8	38.9 ± 2.8
		Rnd.	21.3 ± 2.3	$\mathbf{5.7 \pm 0.5}$	$\mathbf{4.4 \pm 0.3}$
20	1Stp	Rnd.	81.5 ± 4.2	45.0 ± 4.4	25.5 ± 2.8
	UCT	Rnd.	81.5 ± 4.3	44.9 ± 4.7	38.6 ± 3.7
	RIW	NA	83.5 ± 4.8	82.6 ± 5.0	82.8 ± 4.9
		Rnd.	83.4 ± 4.2	39.8 ± 4.2	$\mathbf{10.8 \pm 0.7}$
50	1Stp	Rnd.	230.5 ± 7.8	195.3 ± 11.8	141.5 ± 12.1
	UCT	Rnd.	232.7 ± 7.6	196.7 ± 11.6	175.2 ± 11.9
	RIW	NA	212.9 ± 11.3	215.9 ± 10.8	223.5 ± 9.8
		Rnd.	236.2 ± 6.5	200.4 ± 11.4	$\mathbf{110.6 \pm 11.8}$

Table 2: Same experimental setting as Table 1 over `GridWorld` with a *moving* goal.

difficult to solve using common RL algorithms, such as Q-learning. Langford's two problems allow two actions from every state. The state space for the problems is $S_k = \{i \,|\, 0 \leq i < N\}$ where the number of states value, N, allows the complexity of the problem to be controlled. Langford originally specified the problems as reward-based, here we modify them to be SSP cost-based problems. Reward shaping is commonly used to make Reinforcement Learning easier by encouraging actions, through higher rewards, towards a goal state or states. The first of Langford's problems is named `Antishaping` and uses reward shaping to encourage actions away from the goal state. `Antishaping` has the transition function

$$\mathbf{x}_{k+1} = \begin{cases} \mathbf{x}_k + 1 & \text{if } u_k = 0 \wedge \mathbf{x}_k \notin T_k \\ \mathbf{x}_k - 1 & \text{if } u_k = 1 \wedge \mathbf{x}_k - 1 \geq 0 \wedge \mathbf{x}_k \notin T_k \end{cases} \tag{16}$$

otherwise, the state remains unchanged, $\mathbf{x}_{k+1} = \mathbf{x}_k$. The set containing the goal state is $T_k = \{N-1\}$, which can be achieved by continuously selecting $u_k = 0$. The cost of each transition in `Antishaping` is 0.25 divided by N - x_{k+1}, except when $x_{k+1} = N-1$ where the cost is 0. The problem becomes a large plateau where longer paths become more costly at larger rates. The motivation behind Langford's second problem, `Combolock`, is if many actions lead back to a start state, random exploration is inadequate. The `Combolock` problem has the transition function

$$\mathbf{x}_{k+1} = \begin{cases} \mathbf{x}_k + 1 & \text{if } u_k = \mathbf{sol}_{x_k} \wedge \mathbf{x}_k \notin T_k \\ \mathbf{x}_k & \text{if } \mathbf{x}_k \in T_k \end{cases} \tag{17}$$

otherwise $\mathbf{x}_{k+1}$ is equal to the initial position of 0. The goal state is $T_k = \{N-1\}$ and $\mathbf{sol}_{x_k}$ is either 0 or 1 assigned to state x_k which remains constant. For each state $x \in S$, $\mathbf{sol}_x$ has an equal chance of either being 0 or 1. The cost of each transition in `Combolock` is 1 except for the transition that leads to the terminal state $N-1$ where the cost is 0. While common Reinforcement Learning algorithms such as Q-Learning methods will struggle to solve these domains, it is claimed by Langford that the E^3 (Kearns and Singh 2002) family of algorithms, whose exploration do not rely solely on random policies or reward feedback but on exploring the maximum number of states, will perform well.

Dim.	Alg.	Heu.	Simulator Budget		
			100	1000	10000
10	1Stp	Man.	38.1 ± 2.8	39.0 ± 2.7	38.8 ± 2.7
		Rnd.	43.9 ± 1.9	35.9 ± 2.5	25.3 ± 2.4
	UCT	Man.	37.0 ± 2.9	36.4 ± 2.9	36.4 ± 2.9
		Rnd.	43.8 ± 1.9	38.5 ± 2.5	25.9 ± 1.9
	RIW	Man.	36.4 ± 2.9	36.4 ± 2.9	36.4 ± 2.9
		NA	49.8 ± 0.4	48.8 ± 1.0	49.3 ± 0.8
		Rnd.	44.9 ± 1.7	34.5 ± 2.8	$\mathbf{19.3 \pm 2.1}$
20	1Stp	Man.	76.7 ± 5.5	77.1 ± 5.4	76.7 ± 5.5
		Rnd.	97.9 ± 1.6	88.0 ± 3.4	62.7 ± 4.8
	UCT	Man.	78.0 ± 5.4	78.4 ± 5.3	73.4 ± 5.7
		Rnd.	98.7 ± 1.2	96.4 ± 1.9	77.2 ± 4.2
	RIW	Man.	79.7 ± 4.9	76.7 ± 5.5	76.7 ± 5.5
		NA	100.0 ± 0.0	100.0 ± 0.0	99.6 ± 0.8
		Rnd.	98.5 ± 1.3	88.0 ± 3.4	$\mathbf{29.3 \pm 3.1}$
50	1Stp	Man.	194.6 ± 13.4	191.5 ± 13.6	196.6 ± 13.2
		Rnd.	249.2 ± 1.1	244.4 ± 3.7	216.8 ± 9.3
	UCT	Man.	194.6 ± 13.4	195.6 ± 13.3	184.4 ± 13.9
		Rnd.	249.0 ± 1.9	243.1 ± 4.3	231.6 ± 7.9
	RIW	Man.	208.6 ± 10.7	210.6 ± 11.1	193.5 ± 13.4
		NA	250.0 ± 0.0	250.0 ± 0.0	250.0 ± 0.0
		Rnd.	247.9 ± 2.6	242.9 ± 4.3	196.1 ± 11.3

Table 3: Same settings as Table 1 over `GridWorld` with *fully observable obstacles* and a *stationary* goal.

Methodology

We evaluate the depth-first width-based *rollout* algorithm, RIW(1), with and without being augmented using base policies. $\lambda = 1$ is used for the labels back-propagation. We did not observe statistically significant changes with $\lambda = \infty$. For the `GridWorld` domain we define the features on which RIW(1) plans over as $F = \{(a, i, d) \mid a \in D(\mathbf{x}^i)\}$ where d is the length of the control input path from the initial state, a is the agent's position in the grid in dimension i and $D(\mathbf{x}^i)$ is the domain of the agent's position, a, in dimension i. For `Antishaping` and `Combolock` the feature set will be $F = \{(i, d) \mid i \in N)\}$ where i is the state number the agent is in and N is the number of states of the domain.

Two additional rollout algorithms are also considered, the one-step *rollout* algorithm, RTDP (Barto, Bradtke, and Singh 1995) and the multi-step, selective, *regret* minimisation, *rollout* algorithm Upper Confidence bounds applied to Trees (UCT) (Kocsis and Szepevari 2006). The exploration parameter of UCT is set to 1.0 for all experiments. For all the methods that use a base policies the maximum depth of a simulated trajectory is equal to $H - l$, where l is the depth at which the simulated trajectory began and H is the horizon value of the lookahead. Also, a single, as opposed to multiple, simulated trajectory for the cost-to-go approximation is used, as initial results indicated it is favourable. We also report the algorithms using a Manhattan distance heuristic for the `GridWorld` domains that use obstacles. Using the Manhattan distance for the `GridWorld` problems with obstacles provides a lower bound on the cost-to-go.

Each method on the domains is evaluated at different levels of complexity by varying the number of states. The methods are evaluated using different simulator budgets. The simulator budgets are the maximum simulator calls allowed for the evaluation at each time step. For each algorithm and domain

Dim.	Alg.	Heu.	Simulator Budget		
			100	1000	10000
10	1Stp	Man.	36.6 ± 2.9	35.0 ± 3.0	35.9 ± 2.9
		Rnd.	40.4 ± 2.1	27.2 ± 2.5	15.4 ± 1.6
	UCT	Man.	28.4 ± 2.9	29.5 ± 3.1	18.5 ± 2.7
		Rnd.	41.5 ± 2.0	36.1 ± 2.3	22.2 ± 2.0
	RIW	Man.	28.1 ± 3.0	28.3 ± 3.0	26.5 ± 3.0
		NA	49.5 ± 0.7	49.1 ± 0.9	49.8 ± 0.4
		Rnd.	43.5 ± 1.9	23.4 ± 2.6	$\mathbf{11.1 \pm 1.3}$
20	1Stp	Man.	71.8 ± 5.8	74.0 ± 5.7	71.9 ± 5.8
		Rnd.	97.0 ± 1.7	82.3 ± 3.9	49.8 ± 4.6
	UCT	Man.	53.8 ± 5.7	60.9 ± 6.2	38.0 ± 5.5
		Rnd.	98.0 ± 1.3	87.4 ± 4.0	63.0 ± 4.8
	RIW	Man.	53.5 ± 5.6	$\mathbf{44.1 \pm 5.5}$	44.1 ± 5.6
		NA	100.0 ± 0.0	99.5 ± 1.0	99.5 ± 1.0
		Rnd.	97.6 ± 1.5	79.7 ± 4.4	$\mathbf{20.7 \pm 1.9}$

Table 4: Same settings as Table 1 over `GridWorld` with *partially observable* obstacles and a *stationary* goal.

setting we evaluate the performance over 10 different initial states with 20 episodes per initial state, equalling a total of 200 episodes. The values reported here for each algorithm and domain setting are the average and 95% confidence interval of the costs across the 200 episodes. Each episode was run using a single AMD Opteron 63xx class CPU @ 1.8 GHz, with an approximate runtime of 0.75 seconds per 1,000 simulator calls across the different algorithm and domain settings.

Results are also presented for the Atari-2600 game `Skiing`, which is a SSP problem. We use the OpenAI gym's (Brockman et al. 2016) interface of the Arcade Learning Environment (ALE) (Bellemare et al. 2013) and use the slalom game mode of `Skiing`. In the slalom mode the aim is to ski down the course as fast as possible while going through all the gates. Once the finish line is reached, a 5 second time penalty is applied for each gate that is missed. The reward values provided by ALE for `Skiing` is the negative value of the total time taken plus any time penalties in centiseconds, which we use as a cost. We use the environment settings as described by Machado et al. (2018) with a frame skip of 5 and a probability of 0.25 of repeating the previous action sent to environment instead of the current one, which Machado et al. call sticky actions. For evaluation we use a simulator budget of 100 and partial caching as described by Bandres et al. (2018), in that we cache simulator state-action transitions, thus assuming determinism, but clear the cached transitions when executing an action in the environment. However, the lookahead tree itself is not cleared when executing an action in the environment as is done for the other domains trialed. The maximum episode length is capped at 18,000 frames with a frame skip of 5 this equals 3,600 actions. Using a simulation based cost-to-go approximation is infeasible with a simulator budget of 100 and the maximum episode length of 3,600 actions. Therefore we report the algorithms using a heuristic cost-to-go estimate, which is the the number of gates that have either been missed or are still left times the time penalty of 500 centiseconds. For the RIW(1) algorithms we use the pixel values from the current gray scaled screen at full resolution, that is 180 by 210 pixels, as features.

All experiments were run within the OpenAI gym framework (Brockman et al. 2016) and the code used for the algorithms and domains is available through GitHub [6].

Number of States	Alg.	Heu.	Simulator Budget		
			100	500	1000
10	1Stp	Rnd.	0.6 ± 0.1	0.5 ± 0.1	0.5 ± 0.1
	UCT	Rnd.	0.6 ± 0.1	0.5 ± 0.1	0.5 ± 0.1
	RIW	NA	0.7 ± 0.1	0.7 ± 0.1	0.7 ± 0.1
		Rnd.	0.5 ± 0.1	$\mathbf{0.3 \pm 0.0}$	$\mathbf{0.3 \pm 0.0}$
50	1Stp	Rnd.	1.7 ± 0.1	1.2 ± 0.1	1.1 ± 0.1
	UCT	Rnd.	1.7 ± 0.1	1.3 ± 0.1	1.2 ± 0.1
	RIW	NA	$\mathbf{1.1 \pm 0.0}$	1.1 ± 0.0	1.1 ± 0.0
		Rnd.	1.7 ± 0.1	1.3 ± 0.1	1.1 ± 0.1

Table 5: Average and 95% confidence interval for the cost on `Antishaping`. Costs reported are from 200 episodes over 10 different initial states (20 episodes per initial state). The horizon of each problem is 4 times the number of states.

Results

The different H functions reported here are $H_{\text{NA}} = 0$, the random policy H_{Rnd}, and the Manhattan distance H_{Man}. The algorithms were also evaluated using Knuth's algorithm with a different range of rollouts for the cost-to-go estimate, however, the results are not reported here as they are either statistically indifferent or dominated by the results using H_{Rnd} with a single rollout. Bertsekas (2017) suggests that MCTS algorithms should readily benefit from stronger algorithms to estimate costs-to-go by simulation of stochastic policies. Our experiments showed that if synergies exist these do not manifest when using off-the-shelf stochastic estimation techniques like the ones discussed by Rubinstein and Kroese (2017). Table 1, 2 and 3 report the results of the different lookahead algorithms on the `GridWorld` domain variants with a stationary goal, moving goals and obstacles respectively. For these three domains, results were also collected for a 100x100 grid, however, the results were omitted from the tables as the simulator budgets used were not sufficient to find anything meaningful.

The results on the stationary goal `GridWorld` domain shown in Table 1 provide a number of insights about the *rollout* algorithms reported. First, we can see RIW(1) benefits from using H_{Rnd} rather than H_{NA} where the larger simulator budgets are used. As the simulator budget increases, as could be expected, so does the performance of all the methods using H_{Rnd}. On the contrary, with H_{NA} RIW(1)'s performance remains constant across the different budgets. The explanation for this can be found in the motivating example we gave previously in this paper with the agent preferring the shorter trajectory of driving into the boundary of the grid. Table 1 also shows that given the largest budget and H_{Rnd}, RIW(1) statistically outperforms the other algorithms on the three domains of different size.

Table 2 for `GridWorld` with moving goals has similar patterns as the stationary goal domain in that RIW(1) with H_{Rnd} dominates performance for the largest budget. Also, the majority of performances for the smaller budgets, excluding RIW(1) with H_{NA}, are statistically indifferent.

[6] https://github.com/miquelramirez/width-lookaheads-python

Number of States	Alg.	Heu.	Simulator Budget 100	500	1000
10	1Stp	Rnd.	23.4 ± 2.5	13.5 ± 2.1	10.4 ± 1.7
	UCT	Rnd.	23.6 ± 2.5	12.7 ± 2.0	9.6 ± 1.6
	RIW	NA	27.4 ± 2.6	27.0 ± 2.6	27.9 ± 2.5
		Rnd.	22.9 ± 2.2	$\mathbf{3.6 \pm 0.4}$	$\mathbf{3.6 \pm 0.4}$
50	1Stp	Rnd.	200.0 ± 0.0	196.1 ± 3.8	191.2 ± 5.7
	UCT	Rnd.	199.0 ± 1.9	196.1 ± 3.8	190.2 ± 6.0
	RIW	NA	200.0 ± 0.0	200.0 ± 0.0	199.0 ± 1.9
		Rnd.	199.0 ± 1.9	193.1 ± 5.0	190.2 ± 6.0

Table 6: Same settings as Table 5 over `Combolock`.

`GridWorld` with a stationary goal and obstacles results displayed in Table 3 continues the trend of results. Using the largest budget RIW(1) with H_{Rnd} outperforms all methods on the 10x10 and 20x20 domains. For the 50x50 a number of results are statistically indifferent. For this domain the algorithms using H_{Man} as the base heuristic are also reported. While using the largest budget on the 10x10 grid H_{Rnd} dominates H_{Man}, for the larger 50x50 we see H_{Man} dominates H_{Rnd} for UCT, and is competitive for the other methods.

For the smallest simulator budget on CTP reported in Table 4 using H_{Man} with RIW(1) and UCT are the dominate methods. For the largest simulator budget RIW(1) using H_{Rnd} is dominant over all other methods for both sized domains. We also see that in most cases for the two smaller budgets H_{Man} dominates the H_{Rnd} methods.

Table 5 and 6 show on the smaller 10 state domains RIW(1) with H_{Rnd} is statistically dominant over all other methods on `Antishaping` and `Combolock` for the 500 and 1000 simulator budgets. However, for the more complex 50 state domains, the results of all algorithms using H_{Rnd} are statistically indifferent. It can be observed that using H_{Rnd} with RIW(1) does improve its performance compared with H_{NA} across all the domain settings with simulator budgets of 500 and 1000, besides `Antishaping` with 50 states.

For the `Skiing` Atari-2600 game results in Table 7 H_{Heu} is the heuristic value based on the number of gates missed and remaining as described in the previous section. RIW(1) using H_{Heu} dominates all other methods. Comparing RIW(1) using H_{Heu} results with those reported by Machado et al. (2018), it has similar performance to the DQN algorithm (Mnih et al. 2015) after 100 million frames of training. Since the simulation budget per action we use here is equivalent to 500 frames, and given that the maximum episode duration spans 3,600 actions, RIW(1) achieves the performance in Table 7 considering only 1.8 million frames.

Alg.	Heu.	Simulator Budget 100
1Stp	Heu.	$16{,}524.8 \pm 396.1$
UCT	Heu.	$16{,}220.5 \pm 310.0$
RIW	Heu.	$\mathbf{14{,}222.2 \pm 373.9}$
	NA.	$15{,}854.0 \pm 332.9$

Table 7: Average and 95% confidence interval for the cost on the Atari-2600 `Skiing` game over 100 episodes.

Related Work

Bertsekas (2017) considers AlphaGo Zero (Silver et al. 2017) to be state-of-the-art in MCTS algorithms. It combines the reasoning over confidence intervals first introduced with UCT (Kocsis and Szepevari 2006) and the classic simulation of base policies (Ginsberg 1999), adding to both supervised learning algorithms to obtain, offline, parametric representations of costs-to-go which are efficient to evaluate. The resulting algorithm achieves super-human performance at the game of Go, long considered too hard for AI agents. Rather than using descriptions of game states directly as we do, AlphaZero uses a CNN to extract automatically features that describe spatial relations between game pieces. Like us, AlphaZero's lookahead uses a stochastic policy to select what paths to expand, but rather than Q-factors, uses estimated win probabilities to prioritise controls, and *simulates* the opponent strategy via self-play to generate successor states. Our simulators are always given and remain unchanged, rather than being dynamic as is the case for AlphaZero.

Junyent et al. (2019) have recently presented a hybrid planning and learning approach that integrates Bandres et al. (2018) rollout, with a deep neural network. Similarly to AlphaZero, they use it to both guide the search, and also to extract automatically the set of state features F. Interestingly, Junyent et al.'s work does not use deep neural networks to approximate costs-to-go as AlphaZero does. A significant improvement in performance over Bandres et al. original rollout algorithm is reported with policies learnt after 40 million interactions with the simulator, using an overall computational budget much bigger than the one used by Bandres et al.

Discussion

MCTS approaches typically combine lookaheads and cost-to-go approximations, along with statistical tests to determine what are the most promising directions and focus their sampling effort. The width-based methods described in this paper do so too, but in ways which are, at first sight, orthogonal to existing strategies. It remains an area of active research to map out exactly how the width-based methods described in this paper, and those elsewhere by Junyent et al. (2019) too, provide alternatives to the limitations of existing MCTS approaches. Having said this, there is no general theory guiding the design of MCTS algorithms (Bertsekas 2017), and to avoid generating ad-hoc, problem dependent solutions involuntarily it is important to follow strict protocols that alert of potential lack of statistical significance in results, while relying on a diverse set of benchmarks that can be both easily understood, and highlight limitations of existing state-of-the-art methods and overcome them.

Acknowledgments

This research was supported by the Australian Government Research Training Program Scholarship provided by the Australian Commonwealth Government and the University of Melbourne. The research was also supported by the Defence Science Institute, an initiative of the State Government of Victoria and Google through a Google PhD Travel Scholarship.

References

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite–time analysis of the multiarmed bandit problem. *Machine Learning* (47):235–256.

Bandres, W.; Bonet, B.; and Geffner, H. 2018. Planning with pixels in (almost) real time. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*. AAAI Press.

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Real–time learning and control using asynchronous dynamic programming. *Artificial Intelligence Journal* 72:81–138.

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47:253–279.

Bertsekas, D. P. 2017. *Dynamic Programming and Optimal Control*. Athena Scientific, 4th edition.

Bonet, B., and Geffner, H. 2003. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *Proc. of Int'l Joint Conf. in Artificial Intelligence (IJCAI)*, 1233–1238.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. Openai gym.

Coulom, R. 2006. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games (ICCG)*, 72–83.

Eyerich, P.; Keller, T.; and Helmert, M. 2010. High-quality policies for the canadian traveler's problem. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 51–58.

Geffner, T., and Geffner, H. 2015. Width-based planning for general video-game playing. In *Proc. Artificial Intelligence and Interactive Digital Entertainment Conference*, 23–29.

Ginsberg, M. L. 1999. Gib: Steps toward an expert-level bridge-playing program. In *Proc. of Int'l Joint Conf. in Artificial Intelligence (IJCAI)*, 584–593.

Junyent, M.; Jonsson, A.; and Gomez, V. 2019. Deep policies for width–based planning. In *Proc. of the Int'l Conf. in Automated Planning and Scheduling (ICAPS)*.

Kearns, M., and Singh, S. 2002. Near-optimal reinforcement learning in polynomial time. *Machine learning* 49(2-3):209–232.

Knuth, D. E. 1975. Estimating the efficiency of backtrack programs. *Mathematics of Computation* 29(129):122–136.

Kocsis, L., and Szepevari, C. 2006. Bandit based monte carlo planning. In *Proc. of European Conference in Machine Learning (ECML)*, 282–293.

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

Lipovetzky, N.; Ramirez, M.; and Geffner, H. 2015. Classical planning with simulators: results on the atari video games. In *Proc. of Int'l Joint Conf. in Artificial Intelligence (IJCAI)*, 1610–1616.

Machado, M. C.; Bellemare, M. G.; Talvitie, E.; Veness, J.; Hausknecht, M.; and Bowling, M. 2018. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research* 61:523–562.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518:529.

Papadimitriou, C. H., and Yannakakis, M. 1991. Shortest paths without a map. *Theoretical Computer Science* 84(1):127–150.

Rubinstein, R. Y., and Kroese, D. P. 2017. *Simulation and the Monte Carlo Method*. Wiley & Sons.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; and others. 2017. Mastering the game of Go without human knowledge. *Nature* 550(7676):354.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press, 2nd edition.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. Ff-replan: A baseline for probabilistic planning. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 352–359.

Simplifying Automated Pattern Selection for Planning with Symbolic Pattern Databases

Ionut Moraru[1], **Stefan Edelkamp**[1], **Moises Martinez**[1] and **Santiago Franco**[2]
[1]Informatics Department, Kings College London, UK
[2]School of Computing and Engineering, University of Huddersfield, UK
{firstname.lastname}@kcl.ac.uk, s.franco@hud.ac.uk

Abstract

Pattern databases (PDBs) are memory-based abstraction heuristics that are constructed prior to the planning process which, if expressed symbolically, yield a very efficient representation. Recent work in the automatic generation of symbolic PDBs has established it as one of the most successful approaches for cost-optimal domain-independent planning. In this paper, we contribute two planners, both using *bin-packing* for its pattern selection. In the second one, we introduce a greedy selection algorithm called *Partial-Gamer*, which complements the heuristic given by bin-packing. We tested our approaches on the benchmarks of the last three International Planning Competitions, optimal track, getting very competitive results, with this simple and deterministic algorithm.

1 Introduction

The automated generation of search heuristics is one of the holy grails in AI, and goes back to early work of Gaschnik (1979), Pearl (1984), and Prieditis (1993). In most cases lower bound heuristics are problem relaxations: each plan in the original state space maps to a shorter one in some corresponding abstract one. In the worst case, searching the abstract state spaces at every given search nodes exceeds the time of blindly searching the concrete search space (Valtorta 1984). With pattern databases (PDBs), all efforts in searching the abstract state space are spent prior to the plan search, so that these computations amortize through multiple lookups.

Initial results of Culberson and Schaeffer (1998) in sliding-tile puzzles, where the concept of a pattern is a selection of tiles, quickly carried over to a number of combinatorial search domains, and helped to optimally solve random instances of the Rubik's cube, with non-pattern labels being removed (Korf 1997). When shifting from breadth-first to shortest-path search, the exploration of the abstract state-space can be extended to include action costs.

The combination of several databases into one, however, is tricky (Haslum et al. 2007). While the maximum of two PDBs always yields a lower bound, the sum usually does not. Korf and Felner (2002) showed that with a certain selection of disjoint (or additive) patterns, the values in different PDBs can be added while preserving admissibility. Holte et al. (2004) indicated that several smaller PDBs may outperform one large PDB. The notion of a pattern has been generalized to production systems in vector notation (Holte and Hernádvölgyi 1999), while the automated pattern selection process for the construction of PDBs goes back to the work of Edelkamp (2006).

Many planning problems can be translated into state spaces of finite domain variables (Helmert 2004), where a selection of variables (pattern) influences both states and operators. For disjoint patterns, an operator must distribute its original cost, if present in several abstractions (Katz and Domshlak 2008; Yang et al. 2008).

During the PDB construction process, the memory demands of the abstract state space sizes may exceed the available resources. To handle large memory requirements, symbolic PDBs succinctly represent state sets as binary decision diagrams (Edelkamp 2002). However, there are an exponential number of patterns, not counting alternative abstraction and cost partitioning methods. Hence, the automated construction of informative PDB heuristics remains a combinatorial challenge. Hill-climbing strategies have been proposed (Haslum et al. 2007), as well as more general optimization schemes such as genetic algorithms (Edelkamp 2006; Franco et al. 2017). The biggest issue in this area remains assessing the quality of the PDBs (in terms of the heuristic values for the concrete state space) which can only be estimated. Usually, this involves generating the PDBs and evaluating them (Edelkamp 2014; Korf 1997).

This work contributes by improving the automated pattern selection process. We first define the settings of cost-optimal action planning and give a characterization of a pattern database. We stress spurious states, as they are inevitable to PDB generation. Next, we move to the encoding of the pattern selection problem and how to evaluate the heuristics resulted from them. The main contribution is a greedy partial PDB selection mechanism, which we show that complements well with bin packing, giving close to state of the art results on our benchmarks (bettering the results of the winner of the 2018 International Planning Competition).

2 Background

There are a variety of planing formalisms. Fikes and Nilson (1971) invented the propositional specification language STRIPS, inspiring PDDL (McDermott 1998). Holte and

Hernádvölgyi (1999) invented the production system vector notation (PSVN) for permutation games. Bäckström (1992) prefers the SAS$^+$ formalism, which is a notation of finite-domain state variables over partial states and operators with pre-, (prevail-,) and postconditions.

Definition 1 (SAS$^+$ Planning Task) *is a quadruple $\mathcal{P} = \langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle$, where $\mathcal{V} = \{v_1, \ldots, v_n\}$ is the set of finite-domain variable; $\mathcal{O}$ are the operators which consist of pre-conditions and effects. The remaining two, s_0 and s_* are states. A (complete) state $s = (a_1, \ldots, a_n) \in \mathcal{S}$ assigns a value a_i to every $v_i \in \mathcal{V}$, with a_i in a finite domain D_i, $i = 1, \ldots, n$. For partial states $s^+ \in \mathcal{S}^+$, each $v_i \in \mathcal{V}$ is given an extended domain $D_i^+ = D_i \cup \{\llcorner\}$. We have $s_0 \in \mathcal{S}$ and $s_* \in \mathcal{S}^+$.*

A state space abstraction ϕ *is a mapping from states in the original state space $\mathcal{S}$ to the states in the abstract state space $\mathcal{A}$.*

Let an abstract operator $o' = \phi(o)$ be defined as $pre' = \phi(pre)$, and $post' = \phi(post)$. For planning task described above, the corresponding abstract task is $\langle \mathcal{V}, \mathcal{O}', s'_0, s'_* \rangle$ with $s'_0 \in \mathcal{A}$, $s'_* \in \mathcal{A}^+$, The result of applying operator $o' = (pre', post')$ to an abstract state $a = s'$ satisfying pre', sets $s'_i = post'_i \neq \llcorner$, for all $i = 1, \ldots, n$.

A cost is assigned to each operator. In the context of cost-optimal planning, the aim is to minimize the total cost over all plans that lead from the initial state to one of the goals.

The set of reachable states is generated on-the-fly, starting with the initial state by applying the operators. In most state-of-the-art planners, lifted planning tasks are grounded to SAS$^+$. A STRIPS domain with states being subsets of propositional atoms can be seen as a SAS$^+$ instance with a vector of Boolean variables. The core aspect of grounding is to establish invariances, which minimizes the SAS$^+$ encoding.

Definition 2 (State-Space Homomorphism) *A homomorphic abstraction ϕ imposes that if s' is the successor of s in the concrete state space we have $\phi(s')$ is the successor of $\phi(s)$ in abstract one. This suggests abstract operators $\phi(o)$ leading from $\phi(s)$ to $\phi(s')$ for each $o \in \mathcal{O}$ from s of s'.*

As the planning problem spans a graph by applying a selection of set of rules, the planning task abstraction is generated by abstracting the initial state, the partial goal state *and* the operators. Plans in the original space have counterparts in the abstract space, but not vice verse. Usually, the planning task of finding a plan from $\phi(s_0)$ to $\phi(s_*)$ in $\mathcal{A}$ is computationally easier than finding one from s_0 to s_* in $\mathcal{P}$.

The main issue encountered when working with abstractions are *spurious* paths in the abstract state space that have no corresponding path in the original (concrete) space. An intuitive example of two disconnected paths $s_0 \to s_1 \to s_2 \to s_3 \to \ldots \to s_l$ and $s_{l+1} \to s_{l+2} \to s_{l+3} \to \ldots \to s_m = s_*$, is shown in Figure 1 with $l = 3$. As we map s_l and s_{l+1} to the same abstract state, we have an abstract plan which has no preimage in the original one.

Homomorphic abstractions preserve the property that every path (plan) present in the original space is also present (and shorter) in the abstract state space. Still, abstract operators may yield spurious states.

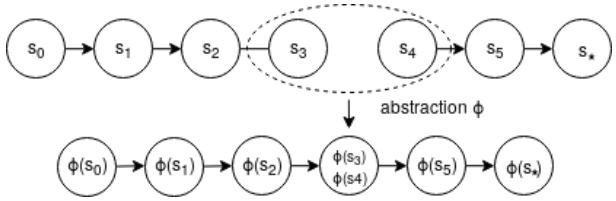


Figure 1: Example of the spurious path problem.

This problem also pops up in unabstracted search spaces. One cause for this is the nature of PDB construction, namely regression search. To illustrate this, consider the (1×3) sliding-tile puzzle with two tiles 1 and 2 and one empty position, the blank. In one SAS$^+$ encoding we have three state variables: two for the position of the tile $t_i \in \{1, 2, 3\}$, $i \in \{1, 2\}$, and one for the position of the blank $b \in \{1, 2, 3\}$. Let $s_0 = (t_1, t_2, b) = (2, 3, 1)$ and $s_* = (1, 2, \llcorner)$. The operators have preconditions $t_i = x$, $b = x + 1$, and effects $t_i = x + 1$, $b = x$, or preconditions $t_i = x$, $b = x - 1$, and effects $t_i = x - 1$, $b = x$, for $i = \{1, 2\}$ and $x \in \{1, 2, 3\}$ (whenever possible). Going backwards from s_*, the planner does not know the location of the blank and beside the reachable state $t_1 = 2$, $t_2 = 3$, $b = 3$ it generates two additional states $t_1 = t_2 = 1$, $b = 2$ and $t_1 = t_2 = 2$, $b = 1$.

How to mitigate the problem? We cannot expect to remove all spurious states, but there is hope to reduce their number. In the case of the sliding-tile puzzle, there is a dual SAS$^+$ encoding with three variables denoting which tile (or blank) is present at a given position p_1, p_2, or p_3. This *exactly-one-of* state invariance is inferred by the static analyzer, but not used in the state encoding. The information, however, can help to eliminate spurious states.

Either spurious paths through abstraction or though regression, they do not affect the lower bound property of the resulting abstraction heuristic. However, they can blow up the PDBs considerably, given that there are abstract states and paths for which no corresponding preimage in the forward space exist. As a result, refined state invariants (including mutex detection of contradicting facts) greatly improve backward search and, thus, reduce the size of pattern databases.

3 Pattern Databases

As planning is a PSPACE-complete problem (Bylander 1994), heuristic search has proven to be one of the best ways to find solutions in a timely manner.

Definition 3 (Heuristic) *A heuristic h is a mapping of the set of states in $\mathcal{P}$ to positive reals $R_{\geq 0}$. A heuristic is called* admissible*, if $h(s)$ is a lower bound of the cost of all goal-reaching plans starting at s. Two heuristics h_1 and h_2 are* additive*, if h defined by $h(s) = h_1(s) + h_2(s)$ for all $s \in \mathcal{S}$, is admissible. A heuristic is* consistent *if for all operators o from s to s' we have $h(s') - h(s) + c(o) \geq 0$.*

For admissible heuristics, search algorithms like A* (Hart, Nilsson, and Raphael 1968) will return optimal plans. If h is also consistent, no states will be reopened during search. This is the usual case for PDBs.

Definition 4 (Pattern Database) *is an abstraction mapping for states and operators* and *a lookup table that for each abstract state a provides the (minimal) cost value from a to the goal state.*

The minimal cost value is a lower bound for reaching the goal of the state that is mapped to a in the original state space. PDBs are generated in a backwards enumeration of the abstract state space, starting with the abstract goal. They are stored in a (perfect) hash table for explicit search, and in the form of a BDD with all abstract states of a certain *h* value while in symbolic search.

Showing that PDBs yield consistent heuristics is trivial (Edelkamp 2014; Haslum et al. 2005), as shortest path distances satisfy the triangular inequality. It has also been shown that for PDBs the sum of heuristic values obtained via *projection* to a disjoint variable set is admissible (Edelkamp 2014). The projection of state variables induces a projection of operators and requires *cost partitioning*, which distributes the cost $c(o)$ of operators *o* to the abstract state spaces (Pommerening et al. 2015). We will discuss more about cost partitioning in section 4.

For ease of notation, we identify a pattern database with its abstraction function ϕ. As we want to optimize PDBs via genetic algorithms, we need an objective function.

Definition 5 (Average Fitness of PDB) *The* average fitness f_a *of a PDB ϕ (interpreted as a set of pairs $(a, h(a))$) is the average heuristic estimate $f_a(\phi) = \sum_{(a,h(a))\in\phi} h(a)/|\phi|$, where $|\phi|$ denotes the size of the PDB ϕ.*

There is also the option of evaluating the quality of PDB based on a sample of paths in the original search space.

Definition 6 (Sample Fitness of PDB) *The fitness f_s of a PDB ϕ wrt. a given sample of (random) paths $\pi_1, \ldots, \pi_m$ and a given candidate pattern selection $\phi_1, \ldots, \phi_k$ in the search space is determined by whether the number of states with a higher heuristic value (compared to heuristic values in the existing collection) exceeds a certain threshold C, i.e.,*

$$\sum_{i=1}^{m} [h_\phi(last(\pi_i)) > \max_{j=1}^{k} \{h_{\phi_j}(last(\pi_i))\}] > C,$$

where $[cond] = 1$, if cond is true, otherwise $[cond] = 0$, and $last(\pi)$ denotes the last state on π.

Definition 7 (Pattern Selection Problem) *is to find a collection of PDBs that fit into main memory, and maximize the average heuristic value*[1].

Definition 8 (Perimeter PDB) *is the result of an unabstracted (blind) backward shortest path search until memory resources are exhausted, setting the value of all yet unreached abstract space to the maximum cost value found in the perimeter, while adding the minimum cost of an operator.*

In several planning tasks, generating the perimeter PDB already solved the problem (Franco et al. 2017).

[1]The average heuristic value has shown empirically that it is a good metric. While it is not the solution to evaluating the pattern selection problem perfectly, it is a good approximation up to this point.

Symbolic Pattern Databases

In symbolic plan search, we encode each variable domain D_j of the SAS$^+$ encoding, $j = 1, \ldots, n$, in binary. Then we assign a Boolean variable x_i to each i, $0 \leq i < \lceil \log_2 |D_1| \rceil + \ldots + \lceil \log_2 |D_n| \rceil$. This eventually results in a characteristic function $\chi_S(x)$ for any set of states S. The ordering of the variables is important for a concise representation, so that we keep finite domain variables as blocks and move inter-depending variables together. The optimization problem of finding such best linear variable arrangement among them is NP-hard. It is also possible to encode operators as Boolean functions $\chi_o(x, x')$ and to progress (and regress) a set of states to accelerate this (pre)image, the disjunction of the individual operators images could be optimized. For action costs, always expanding the set attached to the minimum cost value yields optimal results (Edelkamp 2002). As symbolic search is available for partial states (which denote sets of states), both the forward and the backward symbolic exploration in plan space become similar.

There has been considerable effort to show that PDB heuristics can be generated symbolically and used in a symbolic version of A* (Edelkamp 2002). The concise representation of the Boolean formula for these characteristic functions in a binary decision diagram (BDD) is a technique to reduce the memory requirement during the search. Frequently, the running time for the exploration often reduces as well.

4 Pattern Selection and Cost Partitioning

Using multiple abstraction heuristics can lead to solving more complex problems, but to maintain optimality, we need to distribute the cost of an operator among the abstractions. One way of doing this is present in (Seipp and Helmert 2018). Saturated Cost Partitioning (SCP) has shown benefits to simpler cost partitioning methods. Given an ordered set of heuristics, in our case PDBs, SCP relies on only using those costs which each heuristic uses to create an abstract plan. The remaining costs are left free to be used by any subsequent heuristic. However, considering the limited time budget, this approach is more time consuming compared to other cost partitioning methods (Seipp, Keller, and Helmert 2017).

One such method is 0/1 cost partition, which zeroes any cost for subsequent heuristics if the previous heuristic has any variables affected by that operator. Both SCP and 0/1 allow heuristics values to be added admissibly. SCP dominates 0/1 cost partitioning (given a set of patterns and enough time, SCP would produce better heuristic values), but it is much more computationally expensive than 0/1 cost partitioning.

Franco et al., (2017) shows that, in order to find good complementary patterns, it is beneficial to try as many pattern collections as possible. As such, we implemented 0/1 cost partitioning in our work. We tested using the canonical cost partitioning (Haslum et al. 2007) method as well whenever we added a new PDB, but this resulted in a very pronounced slow down which increased the more PDBs have

already been selected. This was the reason we adopted a hybrid combination approach, where 0/1 cost partition is used on-the-fly to generate new pattern collections, and, only after all interesting pattern collections have been selected, we run the canonical combination method, slightly extended to take into account that each pattern has its own 0/1 cost partition.

Given a number of PDBs in the form of pattern collections (sets of individual patterns, each associated with a cost partitioning function), *canonical pattern databases* will select the best admissible combination of PDB maximization and addition. The computation of the canonical PDB is still expensive, so we execute it only once, right before search starts.

There are many alternatives for automated pattern selection based on bin packing such as random bin packing (PBP), causual dependency bin packing (CBP), which could be refined by a genetic algorithm (Franco et al. 2017).

Greedy Selection

Franco et al. (2017) compared the pattern selection method to the one of Gamer (Kissmann and Edelkamp 2011), which tries to construct one single best PDB for a problem. Its pattern selection method is an iterative process, starting with all the goal variables in one pattern, where the causally connected variables who would most increase the average h value of the associated PDB are added to the pattern.

Following this work, we devised a new *Gamer-style* pattern generation method, which behaves similarly, but which adds the option of *partial pattern database* generation to it. By partial we mean that we have a time and memory limit for building each PDB. If the PDB building goes past this limit, we truncate it in the same way we would do with a perimeter PDB, i.e., any unmapped real state has the biggest h value the PDB building was at when it was interrupted.

An important difference with the Gamer method is that we do not try every possible pattern resulting of adding a single causally connected variable to the latest pattern.

Algorithm 1 Greedy PDBs Creation

function GREEDYPDBS(M,T,S_{min},S_{max},EM) :
Require: time and memory limits T and M, min and max PDB size S_{min} ad S_{max}, evaluation method EM.
  $SelPDBs \leftarrow \emptyset$
  $\mathcal{P}_{sel} \leftarrow \mathcal{P}_{sel} \cup Packer(FFD, S_{min}, M, T, EM)$
  $\mathcal{P}_{sel} \leftarrow \mathcal{P}_{sel} \cup Packer(FFI, S_{min}, M, T, EM)$
  $\mathcal{P}_{sel} \leftarrow \mathcal{P}_{sel} \cup PartialGamer(M, T, EM)$
  Return $\mathcal{P}_{sel}$
end function

function PACKER($Method$,S_{min}, M, T,EM) :
  $SizeLim \leftarrow S_{min}$
  while $(t < T)$ **and** $(m < M)$ **do**
    GENERATE_$\mathcal{P}$($Method$,$SizeLim$)
    if $EM(\mathcal{P})$ **then**
      $\mathcal{P}_{sel} \leftarrow \mathcal{P}$
    end if
    $Size \leftarrow Size * 10$
  end while
  Return $\mathcal{P}_{sel}$
end function

function PARTIALGAMER(M, T,$EvalMethod$) :
  $InitialPDB \leftarrow$ all goal variables
  $SelPDBs \leftarrow InitialPDB$
  while $(t < T)$ **and** $(m < M)$ **do**
    generate all $CandidatePatterns$ resulting of adding one casually connected variable to latest $P \in \mathcal{P}_{sel}$
    for all $P \in CandidatePatterns$ **do**
      if $EM(\mathcal{P})$ **then**
        $\mathcal{P}_{sel} \leftarrow P$
        break
      end if
    end for
  end while
  Return $\mathcal{P}_{sel}$
end function

Genetic Algorithm Selection

A *genetic algorithm* (GA) is a general optimization method in the class of *evolutionary strategies* (Holland 1975). It refers to the recombination, selection, and mutation of *genes* (states in a state-space) to optimize the *fitness* (objective) function. In a GA, a population of candidate solutions is sequentially evolved to generate a better performing population of solutions, by mimicking the process of evolution. Each candidate solution has a set of properties which can be mutated and recombined. Traditionally, candidate solutions are bitvectors, but there are strategies that work on real-valued state vectors.

An early approach for the automated selection of PDB variables by Edelkamp (2006) employed a GA with genes representing state-space variable patterns in the form of a 0/1 matrix G, where $G_{i,j}$ denotes that state variable i is chosen in PDB j. Besides flipping and setting bits, mutations may also add and delete PDBs in the set.

The PDBs corresponding to the bitvectors in the GA have to fit into main memory, so we have to restrict the generation of offsprings to the ones that represent a set of PDB that respect the memory limitation. If time becomes an issue, we stop evolving patterns and invoke the overall search (in our case progressing explicit states) eventually. An alternative, which sometimes is applied as a subroutine to generate the initial population for the GA, is to use bin packing.

Bin Packing

The bin packing problem (BPP) is one of the first problems shown to be NP-hard (Garey and Johnson 1979). Given objects of integer size $a_1, \ldots, a_n$ and maximum bin size C, the problem is to find the minimum number of bins k so that the established mapping $f : \{1, \ldots, n\} \rightarrow \{1, \ldots, k\}$ of objects to bins maintains $\sum_{f(a)=i} a \leq C$ for all $i \leq k$. The problem is NP-hard in general, but there are good approximation strategies such as first-fit and best-fit decreasing (being at most 11/9 off the optimal solution (Dósa 2007)).

In the PDBs selection process, however, the definition of the BPP is slightly different. We estimate the size of the PDB by computing the product (not the sum) of the variable domain sizes, aiming for a maximum bin capacity M imposed by the available memory, and we find the minimum number of bins k, so that the established mapping f of objects to bins maintains $\prod_{f(a)=i} a \leq M$ for all $i \leq k$. By taking the logs on both sides, we are back to sums, but the sizes become fractional. In this case, $\prod_{f(a)=i}$ is an upper bound on the number of abstract states needed.

Taking the product of variable domain sizes is a coarse upper bound. In some domains, the abstract state spaces are much smaller. Bin packing chooses the memory bound on each individual PDB, instead of limiting their sum. Moreover, for symbolic search, the correlation between the cross product of the domains and the memory needs is rather weak. However, because of its simplicity and effectiveness, this form of bin packing currently is chosen for PDB construction.

By limiting the amount of optimization time for each BPP, we do not insist on optimal solutions, but we want fast approximations that are close-to-optimal. Recall, that suboptimal solutions to the BPP do not imply suboptimal solutions to the planning problem. In fact, *all* solutions to the BPP lead to admissible heuristics and therefore optimal plans.

For the sake of generality, we strive for solutions to the problem that do not include problem-specific knowledge but still work efficiently. Using a general framework also enables us to participate in future solver developments. Therefore, in both of the approaches we present in this paper, we focus on the first-fit algorithm.

First-Fit Increasing (FFI), or Decreasing (FFD), is a fast on-line approximation algorithm that first sorts the objects according to their sizes and, then, starts placing the objects into the bins, putting an object to the first bin it fits into. In terms of planning, the variables are sorted by the size of their domains in increasing/decreasing order. Next, the *first* variable is chosen and packed at the same bin with the rest of the variables which are related to it if there is space enough in the bin. This process is repeated until all variables are processed.

5 Symbolic PDB Planners

Based on the results from (Franco et al. 2017), we decided to work only with Symbolic PDBs. Further experiments suggested that PDBs heuristic performs well when it is complemented with other methods. One good combination was using our method to complement a symbolic perimeter PDB, method that we used in the first of the planners we present. The selected method to be complemented first generates a symbolic PDB up to a fixed time limit and memory limit. One advantage of seeding our algorithm with such a perimeter search is that if there is an easy solution to be found in what is basically a brute force backwards search, we are finished before even creating a PDB. Secondly, we combined the Partial-Gamer with bin packing and saw very good results in how they complemented each other. In Figure 2 we see that each method gives good results on their own, Bin-

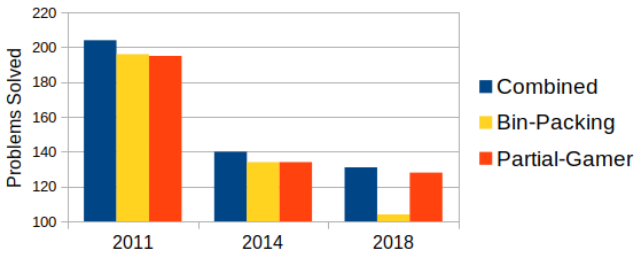


Figure 2: Coverage of Bin Packing, Partial Gamer and of both combined on three latest cost-optimal IPC benchmark problems.

Packing solving 434 and Partial-Gamer 457, but when used together they increase to 475.

In our work, however, we decided to use a hybrid, keeping the forward exploration explicit-state, and the PDBs generated in the backward exploration symbolic. Lookups are slightly slower than in hash tables, but they are still in time linear to the bitvector length.

In this section, we will present two symbolic planners, Planning-PDBs and GreedyPDB, based on the Fast-Downward planning framework (Helmert 2006). The two differ in the pattern selection methods that we use in each of them.

GreedyPDB

We encountered that greedily constructed PDBs outperform the perimeter PDB, which we decided not to use. The two construction methods do not complement well, on the extreme case greedy PDBs will build a perimeter PDB after adding all the variables. There is a significant amount of overlapping between both methods. The collection of patterns received from bin packing, however, complements well the greedily constructed PDBs. One reason for this is that in domains amenable to cost-partitioning strategies, i.e. alternative goals are easily parallelized into a complementary collection of PDBs, bin packing can do significantly better than the single PDB approach. Evaluation is based on the definition of sample fitness. The sample is redrawn each time an improvement was found.

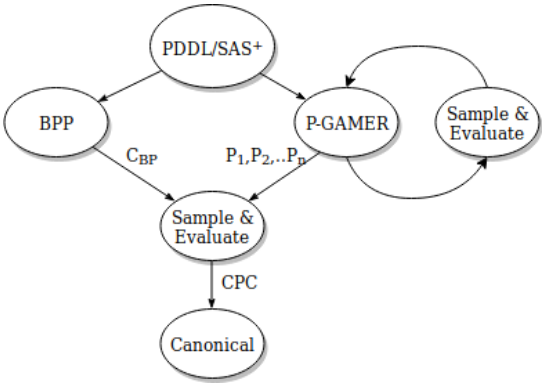


Figure 3: High level architecture of GreedyPDB

Algorithm 1 shows how Greedy PDBs combines two bin packing algorithms with a greedy selection method called

Domain/Method	Agr	Cal	DN	Nur	OSS	PNA	Set	Sna	Spi	Ter	Total
GreedyPDB	13	**12**	**14**	**15**	**13**	16	8	13	11	16	**131**
BP-PDB	6	12	14	12	**13**	**19**	8	11	12	16	123
Scorpion	1	**12**	**14**	12	**13**	0	**10**	13	**15**	14	104
SymBiDir	**14**	9	12	11	**13**	**19**	8	4	6	**18**	114
Complementary1	10	11	14	12	12	18	8	11	11	16	123
Complementary2	6	**12**	13	12	**13**	18	8	**14**	12	16	124
Oracle	14	12	14	15	13	19	10	14	15	18	142

Table 1: Coverage of PDB-type planners on the 2018 International Planning Competition for cost-optimal planning

Partial Gamer. The two bin packing algorithms use First Fit Decreasing (*FFD*) and First Fit Increasing (*FFI*), same used in Planning-PDBs. For FFD we set a limit of 50 seconds, while for FFI we used a limit of 75 seconds (both limits were found empirically to give the best results). To evaluate (EM) if the generated pattern collections should be added to our selection ($\mathcal{P}_{sel}$), we used as an evaluation method a random walk. If enough of the sampled states heuristic values are improved, the pattern is selected. Partial Gamer greedily grows the largest possible PDB by adding causally connected variables to the latest added pattern. If a pattern is found to improve, as defined by the evaluation method, then we add it to the list of selected pattern collections as a pattern collection with a single PDB. Note that we are using symbolic PDBs with time limits on PDB construction, hence a PDB which includes all variables of a smaller PDB does not necesarily dominate it since the smaller PDB might reach a further depth.

An important difference with the Gamer method is that we do not try every possible pattern resulting of adding a single causally connected variable to the latest pattern. As soon as a variable is shown to improve the pattern, we add it and restart the search for an even larger improving pattern. We found this to work better with the tight time limits required by combining several approaches. All the resulting pattern database collections are combined by simply maximizing their individual heuristic values. The PDBs inside each collection were combined using zero-one cost partitioning. The rationale behing the algorithm is that some domains are more amenable to using several patterns where costs are distributed between each patterns, while other domains seem to favour looking for the best possible single pattern.

Planning-PDBs

In *Planning-PDBs*[2], we start with the construction of the perimeter PDB, and continue by using two bin-packing methods to create a collection of PDBs. The first method uses first-fit increasing, while the second being first-fit decreasing. Bin-packing for PDBs creates a small number of PDBs which use all available variables. Even though reducing the number of PDBs used to group all possible variables does not guarantee a better PDB, by having a smaller PDB collections, it is less likely to miss interactions between variables due to them being placed on different PDBs. The bin packing algorithms used ensures that each PDB has a least one goal variable.

If no solution is found after the perimeter PDB has been finished, the method will start generating pattern collections stochastically until either the generation time limit or the overall PDB memory limit are reached. We then decide whether to add a pattern collection to the list of selected patterns if it is estimated that adding such PDB will speed up search. We optimize the results given by the bin-packing algorithm giving it to a GA. It then resolve operator overlaps in a 0/1 cost partitioning. To evaluate the fitness function, the corresponding PDBs is built —a time-consuming operation, which nevertheless payed off in most cases. Once all patterns have been selected, the resulting canonical PDB combination is used as an admissible heuristic to do A* search.

6 Experiments

Following is an ablation-type study were we analyze which components worked best. We run different configurations on the competition benchmarks on our cluster that utilized Intel Xeon E5-2660 V4 with 2.00GHz processors. We compare GreedyPDB and Planning-PDBs with other pattern database and symbolic planners that competed in the 2018 International Planning Competition in the most prestigious and attended deterministic cost-optimal track.

Year/Method	98-09	2011	2014	2018	Total
GreedyPDB	665	**204**	140	**131**	1140
Planning-PDB	678	190	131	123	1122
Scorpion	**785**	190	118	104	**1197**
SymBiDir	686	174	129	114	1064
Comp1	680	185	111	123	1099
Comp2	686	**204**	**155**	124	1169
Oracle	820	227	171	143	1361

Table 2: Overall coverage of PDB-type planners across different International Planning Competitions for cost-optimal planning. All benchmark sets are complete except for the 98-09, in which we use 31 of the domains

Looking at the results of various cost-optimal planners across all domains from the IPC competitions from 1998 to 2018 in Table **??**, we get a good overall picture on the PDB planner performance. Symbolic bidirectional, the benchmark planner in the IPC18 (1064 problems solved overall, 412 for the last 3 IPCs) is almost on par with Scorpion (412) and Complementary1 (419) on the last 3 IPCs, but when adding the 98-09 domains it falls last compared with all

[2]This planner has competed in the 2018 IPC on the Optimal track (Martinez et al. 2018) - https://tinyurl.com/PlanningPDBs

Planner/ Domain	Greedy PDB	Greedy PDB BinPack	Greedy PDB PartGamer	Planning PDB	Scorpion	SymBiDir	Comp1	Comp2	Oracle
Agr	13	4	8	6	1	**14**	10	6	14
Cal	**12**	**12**	**12**	**12**	**12**	9	11	**12**	12
DN	**14**	**14**	12	**14**	**14**	12	**14**	13	14
Nur	14	12	**16**	12	12	11	12	12	16
OSS	**13**	**13**	**13**	**13**	**13**	**13**	12	**13**	13
PNA	18	6	17	**19**	0	**19**	18	18	19
Set	9	9	9	8	**10**	8	8	8	10
Sna	12	11	**14**	11	13	4	11	**14**	14
Spi	11	11	11	12	**15**	6	11	12	15
Ter	15	12	16	16	14	**18**	16	16	18
Bar14	3	3	4	3	3	**6**	3	3	6
Cave14	**7**	**7**	**7**	6	**7**	**7**	**7**	**7**	7
Child14	0	0	0	**5**	0	4	0	1	5
City14	10	10	10	11	14	**18**	10	13	18
Fl14	**20**	**20**	**20**	**20**	8	**20**	14	**20**	20
GED	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	20
Hike	17	17	16	12	10	10	10	**19**	19
Mai	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	5
OS14	8	3	8	5	2	8	5	**13**	13
Pa	4	4	4	3	**6**	2	3	4	6
Tet14	11	11	12	**14**	13	10	11	13	14
TB14	**13**	11	12	5	7	3	7	**13**	13
Tr14	9	9	9	9	**10**	9	9	9	10
Va14	13	14	10	14	13	7	7	**15**	15
Bar11	7	8	8	8	7	**9**	8	8	9
Elev	19	19	19	19	19	**20**	19	19	20
Floor	**12**	**12**	**12**	**12**	6	**12**	**12**	**12**	12
Mys	**20**	**20**	**20**	14	14	11	14	14	20
OS	19	16	16	13	14	14	18	**20**	20
PP	16	16	16	18	**20**	14	18	18	20
Pa	1	1	1	4	**7**	1	1	1	7
Peg	**20**	**20**	**20**	16	17	17	16	19	20
Scan	9	9	9	8	**12**	8	7	9	12
Sok	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	20
TB	**17**	15	15	12	13	9	13	**17**	17
Tr	11	11	11	**13**	**13**	10	10	11	13
Vis	15	16	16	14	8	9	10	**17**	17
Wo	18	13	13	19	**20**	**20**	19	19	20

Table 3: Complete coverage (total number of problems solved) on all of the domains from the previous 3 IPC, cost-optimal track (2011, 2014 and 2018). Domain names have been abbreviated. The planners tested are: three versions of GreedyPDB (one only using Bin Packing, one only using Partial Gamer, and one with both approaches combined); BP-PDB planner; Scorpion and both versions of Complementary planners from IPC 2018; SymBiDir (benchmark planner from IPC 2018).

the others. Scorpion is the overall best in term of instances solved (1197) being by far the best on the older benchmarks. Complementary2 solves a close number of instances 1169, with GreedyPDB close behind with 1140 instances solved.

The reason for the swing in problems solved pre-2011 in favour of the approach Scorpion implements is due to the nature of the domains from that time, most of them catering towards explicit planning. It is also noteworthy that most domains in 2011-2018 benchmarks have 20 instances, while the pre-2011 are on average of 35, with some getting to 202.

By normalizing per domain, we get a slightly different picture, seen in Table 4. As there are some repeating domains in the benchmark sets from different IPCs, we insist on showing the results split over different IPCs, which are meant to encourage domain-independent planning.

On the 2018 benchmark, likely the most challenging one featuring a wide range of expressive application domain models, GreedyPDB would have actually won the competition (Table 1). This indicates that for several planning problems, the best option is to keep growing one PDB with the

	Problems Solved	Coverage	Normalized Coverage
GreedyPDBs	1140	55.04%	61.08%
Planning-PDBs	1122	54.17%	59.42%
Scorpion	**1197**	**57.79%**	59.26%
Sym-BiDir	1053	50.84%	55.46%
Complementary1	1099	53.06%	57.60%
Complementary2	1164	56.15%	**62.08%**

Table 4: Results as number of problems solved, coverage and normalized coverage.

greedy pattern selector, and compare and merge the results with a PDB collection based on bin packing [3].

7 Related Work

Pattern Databases have become very popular since the 2018 International Planning Competition showed that top five planners employed the heuristic in their solver. However, the topic has been vastly researched prior to this competition, a lot of work going in the automated creation of a PDB, with the best know being the iPDB of Haslum et al., (2007) and the GA-PDB by Edelkamp (2006). The first performs a hill-climbing search in the space of possible pattern collections, while the other employs a bin-packing algorithm to create initial collections, that will be used as an initial population for a genetic algorithm. iPDB evaluates the patterns by selecting the one with the higher *h*-value in a selected sample set of states, while the GA of the GA-PDB uses the average heuristic value as its fitness function.

Another two approaches related to our work is Gamer (Kissmann and Edelkamp 2011) and CPC (Franco et al. 2017). The first is in the search of only one best PDB, starting with all the goal variables, and adding the one that it will increase the average heuristic value. CPC is a *revolution* of the GA-PDB approach, aiming to create pattern collections with PDBs that are complementary to eachother. It also employes a GA and its evaluation is based on Stratified Sampling.

8 Conclusion and Discussion

The 2018 International Planning Competition in cost-optimal planning revealed that symbolic PDB planning probably is the best non-portfolio approach. In fact, five of the top six IPC planners were based on heuristic search with PDB and/or symbolic search, while the winning portfolio used such a planner (namely SymBA*, the winner of IPC 2014) for more than half of its successful runs.

In this paper, we present two methods building on top of the CPC approach by Franco et al., (2017), one incremental on an existing work (Planning-PDB), and one that is a reformulation of how it creates complementary pattern collections (GreedyPDB), by combining it with an adapted version of the Gamer approach (Kissmann and Edelkamp 2011). In both we have only one bin-packing solver, removing the multi-armed bandit algorithm to select its packing algorithm. In GreedyPDB, we also removed the optimization done with a GA over the pattern collections, seeing that bin-packing and partial-gamer complement already very well each other. Overall, the structure of GreedyPDB in comparison with CPC is very much simplified, with a small loss of coverage on the problem set of the IPC 2014.

Using different pattern generators to complement the two seeding heuristics was extremely successful. It improved our overall results for all the methods we tested compared to simply using the seeding heuristics. One of the best performing method is the combination of an incremental pattern selection with advanced bin packing. When combining both pattern selection methods, the results are greatly improved, and GreedyPDB would have won the last IPC even ahead of the best portfolio planners (solving 5 more problems), thus contributing a new state-of-the-art in cost-optimal planning.

It is probable that using SCP instead of canonical would improve results. It is also likely that if we used SCP online, i.e., for evaluating whether to add a PDB to the current selected set, instead of the current 0/1 approach a PDB is evaluated, would significantly reduce the total number of patterns we can try given the IPC time limit. How to navigate the trade-off between SCP's better heuristic values vs 0/1's faster computational time is future research.

However, as seen with the impressive results of Complementary2 in the 2011 and 2014 competition benchmark, there is no free lunch. Which pattern generator method is best depends on the benchmark domain it is applied to. By the obtained diversity in the individual solutions, an oracle deciding which pattern selector to take would have solved more problems, so that a portfolio planner could exploit this.

[3] We include all the results of our experiments IPC11-18 in Table 3. The rest are available online.

References

[Bäckström 1992] Bäckström, C. 1992. Equivalence and tractability results for sas+ planning. In *KR*, 126–137.

[Bylander 1994] Bylander, T. 1994. The computational complexity of propositional strips planning. *Artificial Intelligence* 69(1-2):165–204.

[Culberson and Schaeffer 1998] Culberson, J. C., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(4):318–334.

[Dósa 2007] Dósa, G. 2007. The tight bound of first fit decreasing bin-packing algorithm is ffd (i) 11/9opt (i)+ 6/9. In *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*. Springer. 1–11.

[Edelkamp 2002] Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *AIPS*, 274–283.

[Edelkamp 2006] Edelkamp, S. 2006. Automated creation of pattern database search heuristics. In *International Workshop on Model Checking and Artificial Intelligence*, 35–50. Springer.

[Edelkamp 2014] Edelkamp, S. 2014. Planning with pattern databases. In *Sixth European Conference on Planning*.

[Fikes and Nilsson 1971] Fikes, R. E., and Nilsson, N. J. 1971. Strips: A new approach to the application of theo-

rem proving to problem solving. *Artificial intelligence* 2(3-4):189–208.

[Franco et al. 2017] Franco, S.; Torralba, A.; Lelis, L. H.; and Barley, M. 2017. On creating complementary pattern databases. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 4302–4309. AAAI Press.

[Garey and Johnson 1979] Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractibility, A Guide to the Theory of NP-Completeness*. Freeman & Company.

[Gaschnig 1979] Gaschnig, J. 1979. A problem similarity approach to devising heuristics: First results. 434–441.

[Hart, Nilsson, and Raphael 1968] Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4(2):100–107.

[Haslum et al. 2005] Haslum, P.; Bonet, B.; Geffner, H.; et al. 2005. New admissible heuristics for domain-independent planning. In *AAAI*, volume 5, 9–13.

[Haslum et al. 2007] Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. 1007–1012.

[Helmert 2004] Helmert, M. 2004. A planning heuristic based on causal graph analysis. 161–170.

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

[Holland 1975] Holland, J. 1975. *Adaption in Natural and Artificial Systems*. Ph.D. Dissertation, University of Michigan.

[Holte and Hernádvölgyi 1999] Holte, R. C., and Hernádvölgyi, I. T. 1999. A space-time tradeoff for memory-based heuristics. In *AAAI/IAAI*, 704–709. Citeseer.

[Holte et al. 2004] Holte, R.; Newton, J.; Felner, A.; Meshulam, R.; and Furcy, D. 2004. Multiple pattern databases. 122–131.

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

[Kissmann and Edelkamp 2011] Kissmann, P., and Edelkamp, S. 2011. Improving cost-optimal domain-independent symbolic planning. In *Twenty-Fifth AAAI Conference on Artificial Intelligence*.

[Korf and Felner 2002] Korf, R. E., and Felner, A. 2002. *Chips Challenging Champions: Games, Computers and Artificial Intelligence*. Elsevier. chapter Disjoint Pattern Database Heuristics, 13–26.

[Korf 1997] Korf, R. E. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. 700–705.

[Martinez et al. 2018] Martinez, M.; Moraru, I.; Edelkamp, S.; and Franco, S. 2018. Planning-pdbs planner in the ipc 2018. *IPC-9 planner abstracts* 63–66.

[McDermott 1998] McDermott, D. 1998. The 1998 ai planning systems competition. In *AI Magazine*, 35–55.

[Pearl 1984] Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

[Pommerening et al. 2015] Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. 2015. From non-negative to general operator cost partitioning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

[Preditis 1993] Preditis, A. 1993. Machine discovery of admissible heuristics. *Machine Learning* 12:117–142.

[Seipp and Helmert 2018] Seipp, J., and Helmert, M. 2018. Counterexample-guided cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research* 62:535–577.

[Seipp, Keller, and Helmert 2017] Seipp, J.; Keller, T.; and Helmert, M. 2017. A comparison of cost partitioning algorithms for optimal classical planning. In *Twenty-Seventh International Conference on Automated Planning and Scheduling*.

[Valtorta 1984] Valtorta, M. 1984. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences* 34:48–59.

[Yang et al. 2008] Yang, F.; Culberson, J.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research* 32:631–662.

Simultaneous Re-Planning and Plan Execution for Online Job Arrival

Ivan Gavran[1], **Maximilian Fickert**[2], **Ivan Fedotov**[1], **Jörg Hoffmann**[2], **Rupak Majumdar**[1]
[1]Max Planck Institute for Software Systems, Kaiserslautern, Germany
[2]Saarland University, Saarland Informatics Campus, Saarbrücken, Germany
{gavran, ivanan, rupak}@mpi-sws.org, {fickert, hoffmann}@cs.uni-saarland.de

Abstract

In many AI planning applications, an agent receives new jobs (additional non-conflicting goals) while plan execution is still ongoing. Vanilla solutions are to (a) finish execution before tackling the new job, or to (b) interrupt execution and re-plan immediately. Option (a) misses opportunities to smoothly integrate the new job into the plan, while (b) leaves the agent idle during re-planning. We introduce *simultaneous re-planning and execution* (SRE), a planning algorithm that avoids both disadvantages. SRE re-plans for both the old and new jobs while the current plan is still being executed. The key difficulty is that, then, the initial state for the revised plan–the state in which plan execution is at the end of re-planning–depends on the time taken for re-planning. We address this through a variant of A* that starts with several speculative initial states, and incorporates time-aware search information to differentiate between these. On a collection of extended planning competition benchmarks, our algorithm consistently outperforms both (a) and (b).

Introduction

In AI planning (Ghallab, Nau, and Traverso 2004), the task is to find a schedule of actions leading from the initial state of an agent to a goal state. Planning tools are given a description of states, actions, and goal as input, and should automatically produce a plan. While planning is often seen as a discipline of "thinking before acting," many problems require a constant interplay between thinking and acting (Myers 1999; Ghallab, Nau, and Traverso 2016): e.g., Mars rovers (Estlin et al. 2000; Knight et al. 2001), high-speed manufacturing (Ruml, Do, and Fromherz 2005), or modular printer controllers (Ruml et al. 2011). Here, we address *continual online planning* (Lemons et al. 2010; Burns et al. 2012), where an agent has to tackle a changing set of goals. We consider a special case we call *continual online job arrival*, where the new goal is akin to an additional *job*, that does not conflict with the previous goal.

Similar problems have been considered in different circumstances and under different assumptions. Agents with knowledge about the goal arrival distribution can anticipate future goals and plan accordingly (Burns et al. 2012). If the new goals are known to be similar to the old ones, plan repair can be used (Fox et al. 2006). Here we make neither of these assumptions, tackling arbitrary new goals/jobs arriving online.

There are two vanilla solutions for an online plan execution and re-planning loop in our context: (a) keep executing the current plan to its end before starting the execution of the (re-planned) new plan incorporating the new job; or (b) interrupt the execution of the current plan and wait for the re-planning process to finish. Both strategies have pros and cons. Option (a) allows (some of) the re-planning to be done in parallel to the execution. However, while moving towards the old goal, the agent might be moving away from the new goal, thus missing the opportunity to smoothly incorporate the new job into the current plan. Option (b) takes this opportunity, but leaves the agent idle for the entire re-planning process, which is wasteful when re-planning takes a long time. In this paper we direct our attention towards scenarios where the planning time can not be assumed to be negligible with respect to execution time.

We propose a *simultaneous re-planning and executing* algorithm (that we will refer to as *SRE*) that plans for both the old and the new task while executing the current plan, thus combining the advantages of both previously described strategies. This raises a new challenge: the agent changes its state while re-planning, so the initial state for the revised plan depends on the time taken by the re-planning process. That process must thus be aware of, and reason about, its own duration in order to determine the initial state for the revised plan.

Planners able to reason about their own planning time are called *time-aware* (e.g., (Burns, Ruml, and Do 2013; Cashmore et al. 2018)). Unlike classical planners, which optimize plan cost (e.g. the plan duration), time-aware planners use the (estimated) time needed for planning as a part of their cost function. To illustrate: a user cares about the time needed to get a cup of coffee from a robot; what fraction of that time was spent planning, and what fraction was spent executing the plan, is irrelevant to the user. Time-aware planners do not solve our challenge here as they still assume a fixed initial state. However, as we shall see, the techniques used by time-aware planners can help address that challenge in our context.

Our SRE algorithm is a variant of A* (Hart, Nilsson, and Raphael 1968) that takes advantage of time-aware techniques to adapt to the setting with asynchronous task arrivals. There are two high-level differences between SRE and A*.

First, while A* starts its search from a single initial state, SRE starts with a number of potential initial states. These states represent a speculation on the state in which the agent might be once the planning is done. Second, A*'s ordering function is extended with an additional heuristic function, estimation of when the planning process will finish, informing the search about which of the possible initial states, and search paths below these, is promising to explore. We prove that, under suitable conditions on the heuristic functions used by SRE, the first solution it finds is better than any other it might find by continuing the search. This property and its proof correspond to a similar property of the time-aware search algorithm Bugsy (Burns, Ruml, and Do 2013).

Overall, our contributions are

- a time-aware search algorithm for online AI planning with asynchronous task arrival;
- an implementation of the algorithm in Fast Downward (Helmert 2006), leveraging time-aware planning techniques (Dionne, Thayer, and Ruml 2011) as well as heuristic search planning techniques (Hoffmann and Nebel 2001);
- an empirical comparison of the algorithm to two baselines on a collection of benchmarks from the international planning competition (IPC), which we extended for our online setting; SRE consistently outperforms both vanilla solutions.

Problem definition

We formulate our setting as a variant of classical planning with online job arrivals during an ongoing execution. The focus is on the moment when a new task arrives while the agent is already executing its *current plan*. This generalizes straighforwardly to a series of arriving jobs, assuming that they arrive once the planning phase is done.

Background

We consider the *finite-domain representation (FDR)* (Bäckström and Nebel 1995; Helmert 2009) for classical planning tasks:

Definition 1. A **planning task** is a tuple (V, A, c, s_0, s_*):

- V is a finite set of *state variables*, each with a finite domain of possible values,
- A is a finite set of *actions*. Each action a is a pair (pre_a, eff_a) of partial variable assignments called *preconditions* and *effects*,
- $c\colon A \to \mathbb{R}$ is a function assigning a cost to every action,
- s_0 is a complete variable assignment called *initial state*,
- s_* is a partial variable assignment called *goal*.

We denote the set of all complete variable assignments, or *states*, by $\mathcal{S}$. A partial assignment p is said to be *compliant* with a state $s \in S$ (denoted by $p \subseteq s$) if there is no variable in the domain of p to which p and s assign different values. An action $a \in A$ can only be applied to a state $s \in S$ if $pre_a \subseteq s$. The outcome of that application is state $s[\![a]\!]$, that is the same as s, except that the variables in the domain of partial assignment eff_a are changed accordingly.

A solution (*plan*) to a planning task is a sequence of actions $\overline{a_1, a_2, \ldots, a_n}$ with the overall cost $C(\overline{a_1, a_2, \ldots, a_n}) = \sum_{i=1}^{n} c(a_i)$, leading from s_0 to a state compliant with s_*.

In what follows, the costs of actions (function c) will be interpreted as the duration to execute them. We do not, however, consider concurrent plans as in temporal planning (Fox and Long 2003), limiting our focus to sequential plans with action durations instead. Exploring concurrent temporal planning remains an important topic for future work.

We are considering tasks where the set of goals is not fixed, and new goals may appear online. This has been called *continual online planning (COP)* (Lemons et al. 2010; Burns et al. 2012). COP tasks have been defined as Markov Decision Processes where additional goals may arrive at each time step, and world states are extended with the current goal set. We adapt this notion to COP tasks as classical planning tasks that are extended with a second goal condition, assumed to arrive during the execution of the plan for the original goal.

Definition 2. A **continual online planning (COP) task** is a tuple $(V, A, c, s_*^{\text{OLD}}, s_*^{\text{NEW}}, s_0, \pi_{s_0, s_*^{\text{OLD}}})$ where

- V is a finite set of *state variables*, each with a finite domain of possible values.
- A is a finite set of actions. Each action a is a pair (pre_a, eff_a) of partial variable assignments,
- $c\colon A \to \mathbb{R}$ is a function that assigns a cost to every action $a \in A$. We interpret the cost $c(a)$ as the duration needed to execute action a,
- s_*^{OLD} is a partial variable assignment called *old goal*,
- s_*^{NEW} is a partial variable assignment called *new goal*,
- s_0 is the state denoting the agent's position at the time when s_*^{NEW}, the new goal, appeared,
- $\pi_{s_0, s_*^{\text{OLD}}} = \overline{a_1 a_2 \ldots a_n}$ is a sequence of actions, taking the agent from the state s_0 to a state compliant with the old goal s_*^{OLD} (the agent's *current plan*).

A solution to a COP task is a plan π consisting of two parts: a prefix of $\pi_{s_0, s_*^{\text{OLD}}}$ and the newly planned extension. There must exist $1 \leq j \leq n$ such that $\pi = \overline{a_1 a_2 \ldots a_j b_1 \ldots b_m}$ (where the extension, denoted by actions b_1 to b_m, can be empty). The state (partial assignment) to which the plan π takes the agent must be compliant with both s_*^{OLD} and s_*^{NEW}. A solution is said to be optimal if it minimizes the overall planning and execution time, i.e., the time from the arrival of the new job s_*^{NEW} to the end of the execution of π.

Continual Planning for Online Job Arrival

In this work we consider COP tasks with particular properties making the achievement of planning goals arriving online akin to the achieval of "jobs" as in scheduling problems. Namely, (i) executing plans for previous goals should not preclude the possibility to achieve new goals; and (ii) achieving new goals should not necessitate deleting previous ones.

Definition 3. A **continual planning for online job arrival (COJA) task** is a COP task $(V, A, c, s^{\text{OLD}}_*, s^{\text{NEW}}_*, s_0, \pi_{s_0, s^{\text{OLD}}_*})$ with the following two properties.

(i) **recoverable states**: for every state s' reached from a state s with an action sequence $\vec{\alpha}$, there exists an action sequence $\overleftarrow{\alpha}$ so that $s'[\![\overleftarrow{\alpha}]\!]$ agrees with s on all variable values that appear as preconditions or goals in the task.

(ii) **stable goals**: for every state s from which s^{NEW}_* can be achieved, there exists a minimum-cost action sequence α doing so without ever changing the assignment $s \cap s^{\text{OLD}}_*$.

Restriction (i) relates to known notions of invertibility and undoability (e.g. (Hoffmann 2005; Daum et al. 2016)). It serves two purposes. First, it allows to err in the prediction of when re-planning will terminate (and thus what the new initial state will be). If the re-planning takes longer than estimated, then the plan execution will have arrived at a state s_j behind the new initial state s_i used by the new plan. Recoverable states allow to nevertheless use the new plan, by going back from s_j to (a state subsuming) s_i first. Second, (i) alleviates necessary, or accidental, conflicts between the previous goal s^{OLD}_* vs. the new goal s^{NEW}_*. It may, in general, happen that the new plan temporarily deletes s^{OLD}_* (e.g. in the Blocksworld if s^{NEW}_* requires to move a block at the bottom of a stack). Given (i), re-achieving s^{OLD}_* is always possible.

Stable goals (ii) demand that at least one optimal plan for s^{NEW}_* does not delete whichever parts of s^{OLD}_* are already achieved. This restriction is sensible as it excludes *necessary* conflicts between the previous vs. the new goals, i.e., situations where achieving s^{NEW}_* necessarily involves deleting s^{OLD}_*. The optimality requirement makes sure that deleting s^{OLD}_* can be avoided without a cost penalty.

Even with (ii), it may of course happen that parts of the previous plan, executed during re-planning in our approach, have to be un-done later on. Furthermore, the re-planning process may not find a minimum-cost plan, or for other reasons return a plan deleting s^{OLD}_*. Such *accidental* goal conflicts are, however, qualitatively different from the necessary ones excluded by (ii). That said, stable goals are not a strict requirement of our approach, but merely a "nice to have" property. Indeed, one of our benchmark domains does not satisfy (ii).

Many applications have recoverable states and stable goals. Examples include warehouse logistics, abstract encodings of Mars rover control, and various types of manufacturing problems. Hoffmann (2005) specifies syntactic criteria allowing to identify tasks with recoverable states, and, given an action sequence $\vec{\alpha}$, to quickly find the recovery sequence $\overleftarrow{\alpha}$.

In our experiments, we focus on domains where each action has an immediate inverse action, and thus the cost of $\overleftarrow{\alpha}$ equals that of $\vec{\alpha}$. This simplifies matters as, given $\vec{\alpha}$, the cost of the recovery sequence is known exactly. It remains a topic for future work to drop this assumption (e.g. drawing on Hoffmann's criteria as just mentioned).

Algorithm 1 SRE

```
1:  procedure SRE(s_0, s*^OLD, h, π_{s_0,s*^OLD}, s*^NEW, R)
2:      γ ← 0
3:      open ← {(r, r) | r ∈ R}
4:      closed ← ∅
5:      while open ≠ ∅ do
6:          γ ← γ + 1
7:          (s, ref_s) ← arg min_{(m,ref_m)∈open} f(m, ref_m, γ)
8:          if (s*^OLD ∪ s*^NEW) ⊆ s then
9:              return path to s
10:         closed ← closed ∪ {(s, ref_s)}
11:         for m ∈ successors(s) do
12:             ref_m ← ref_s
13:             if ((m, ref_m) ∉ (open ∪ closed) or
                    g(ref_m, m) < g_old(ref_m, m)) then
14:                 open = open ∪ {(m, ref_m)}
15:     return fail
16: f :: (m, ref_m, γ) ↦ g(s_0, ref_m)+
                         g(ref_m, m) + h(m)+
                         overshot(m, ref_m, γ)
```

Simultaneous Re-Planning and Execution

We now describe our simultaneous re-planning and execution algorithm SRE, which is an extension of A* to solve COJA tasks. We first specify the algorithm (and how it relates to A*), then we discuss its theoretical properties.

Algorithm

Algorithm 1 shows the pseudocode of the SRE algorithm. Its structure closely resembles the structure of A*, and the important differences in the pseudocode are highlighted in red.

In contrast to A*, SRE uses a set of potential starting nodes, which we call *reference nodes* and which are given to the algorithm as a parameter R. These starting nodes are different "guesses" on which state the agent will be in when the planning finishes. Each reference node is a potential last state of the current plan towards s^{OLD}_* before deviating from it (the current plan is also given as a parameter $\pi_{s_0, s^{\text{OLD}}_*}$).

The open list (*open*) is initialized using the reference nodes (line 3). Each element of *open* is a pair containing the search node and its corresponding reference node (the node at which it deviates from the original plan). Each newly created search node retains the reference node of its parent (line 12).

Like in A*, nodes in the open list are expanded in a best-first order according to a scoring function f, and put into the closed list afterwards. When a node is expanded, its successors are inserted into the open list if they are new or are reached with a lower g-value than before (line 13).

Line 8 shows the termination condition. Following Definition 2, we must check whether both the original goal s^{OLD}_* and the new goal s^{NEW}_* are achieved.

Finally, the most important difference is the ordering function f for the open list (line 16). The modified f-

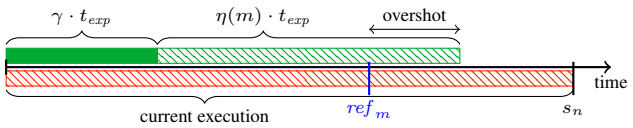
function has three parameters[1]: a node, its reference node, and the number of expansions made by the algorithm so far, γ. The f-function assigns a score to a pair (m, ref_m) based on three parts. The first part is $g(s_0, ref_m)$.[2] It represents the time required to move from the initial node s_0 to the reference node that was used to reach m. The second part is $g(ref_m, m) + h(m)$, the same as A*'s f-function. It represents the time needed to get from the reference node ref_m to m, the node under consideration, combined with the estimate of the time needed to reach the goal from m. The third part depends on the function denoted by $overshot(m, ref_m, \gamma)$. This represents the penalty of having to go back to the correct reference node if the agent has already moved past it before planning finishes. This is possible because in COJA tasks, states are recoverable.

We denote the heuristic function estimating the remaining number of expansions until planning finishes by $\eta(m)$. In order to connect this to the execution time, the number of expansions is multiplied by the time per expansion t_{exp} (Burns, Ruml, and Do 2013).

We define the overshot function with respect to a node m, its reference node ref_m, and the number of expansions so far γ. Let $\vec{\alpha}$ be the subsequence of actions on $\pi_{s_0, s_*^{\text{OLD}}}$ taking the agent from the reference node ref_m to the state in which it would be at time $(\gamma + \eta(m)) \cdot t_{exp}$ if planning is estimated to end after the execution reaches ref_m, and an empty sequence otherwise. Let $\overleftarrow{\alpha}$ be the recovery sequence of $\vec{\alpha}$. The overshot is then defined as $overshot(m, ref_m, \gamma) = C(\vec{\alpha}) + C(\overleftarrow{\alpha}) + \max((\gamma + \eta(m)) \cdot t_{exp} - C(\pi_{s_0, s_*^{\text{OLD}}}), 0)$.

The overshot is 0 if the planning is estimated to finish before reaching the reference node ref_m. Otherwise, it describes the additional execution time incurred by moving past the reference node and back. If planning takes longer than total execution of $\pi_{s_0, s_*^{\text{OLD}}}$, then the agent will additionally have to wait in s_n, the last node of $\pi_{s_0, s_*^{\text{OLD}}}$ (this is described by the last term of the overshot function).

Consider the following illustration:



The red dashed bar denotes the time needed to execute the current plan leading to $s_n \subseteq s_*^{\text{OLD}}$. The green bar labeled by $\gamma \cdot t_{\text{exp}}$ is time spent planning so far and the dashed green bar $(\eta(m) \cdot t_{\text{exp}})$ shows the estimation on when the planning will finish. In the illustration, the planning time is estimated to exceed the time when the selected reference node ref_m is reached. The overshot describes this additional execution time, plus the time it takes to go back to ref_m.

Having γ as an argument for f has an interesting consequence: it now matters when the function f is evaluated for the relative order of the nodes in *open*. In practice, we do not re-evaluate f on all the nodes in the open list each time the best element is retrieved (line 7). Instead, we approximate the value of f-function by keeping the search nodes sorted only by $g + h$, but separately for each reference node. Subsequently, we do the full evaluation only to select the next reference node for which a node should be expanded using the nodes with minimal $g + h$ for each reference node. This approximation is justified by the fact that a changed value of γ affects all the nodes corresponding to the same reference node equally. The loss of precision comes from disregarding differences in η.

Coming back to the classical A* formulation, note that there is a parallelism between g and $\gamma \cdot t_{\text{exp}}$ (execution time and planning time so far) as well as between h and $\eta \cdot t_{\text{exp}}$ (estimated time till the end of execution and planning, respectively). There is an important difference though: while exploring a node will not influence the g value of other nodes, γ will change its value for all nodes expanded in the future. Note additionally that the *true value* of function g (usually denoted by g^*) does not depend in any way on heuristic g. In contrast, how many expansions are needed until the end of planning (denoted by η^*) depends on the heuristic function η.

Theoretical Analysis

For A*, it can be shown that the algorithm finds an optimal solution, provided that the heuristic function is admissible (and nodes can be reopenend). A similar guarantee can not be given for SRE. The essential difference between the two settings (and thus necessarily between the two algorithms) is that for a classical planning task, the exploration of the state space during the planning phase comes at no cost. On the other hand, in an online setting, exploring a part of the search space that is not going to be used in the solution can decrease the quality of the final plan, since that time was not used effectively. Therefore, unless the heuristic functions η and h were perfect, there is no guarantee that SRE will find an optimal solution. We are, however, able to prove that SRE's stopping policy is the correct one. Moreover, in this section, we revisit the baselines mentioned in the introduction and analyze the circumstances under which they can outperform SRE.

SRE stops the search as soon as the first state compliant with both of its goals is found, which raises the question if there is some trade-off between continuing the search and the quality of the solution. We show that continuing the search can not result in a better plan, assuming the heuristic functions h and η are admissible.

We will use $h^*(m)$ to denote the true value of the cost to reach the goal from m, and $\eta^*(m)$ to denote the number of expansions from node m to the end of planning. Following the same notation style, $f^*(m, ref_m, \gamma_m)$, and $overshot^*(m, ref_m, \gamma_m)$ denote functions f and $overshot$ calculated using $h^*(m)$ and $\eta^*(m)$ instead of the heuristics h and η. We are using the notation $\gamma = \gamma_m$ to indicate that the third argument of the f-function is the value of γ when the node m was explored.

Theorem 1. *Let h be admissible with respect to planned execution time and η admissible with respect to number*

[1] s_0 is treated as a default parameter

[2] In SRE the g-function takes two arguments, and returns the cost (time in our context) from the first to the second. In A* this is implicit as it is only used to denote the cost from the initial node.

of expansions. Additionally, assume that for the path α that is a prefix of the path α' it holds $C(\vec{\alpha}) + C(\overleftarrow{\alpha}) \leq C(\vec{\alpha'}) + C(\overleftarrow{\alpha'})$ (well-behaved recovery paths). Let $\sigma_1 = \overline{s_0 s_1 \dots s_i p_1 p_2 \dots p_m}$ be the sequence of states corresponding to the first solution π_1 found by SRE (s_i is the reference node and p_m is the final state of σ_1). Assume the algorithm continued the search and found another solution, with its sequence of states being $\sigma_2 = \overline{s_0 s_1 \dots s_j q_1 q_2 \dots q_n}$ (s_j is the reference node and q_n is the final state of σ_2). It holds that $f^*(p_m, s_i, \gamma_{p_m}) \leq f^*(q_n, s_j, \gamma_{q_n})$

Proof.

$$
\begin{aligned}
& f^*(p_m, s_i, \gamma_{p_m}) = \\
& = g(s_0, s_i) + g(s_i, p_m) + overshot^*(p_m, s_i, \gamma_{p_m}) \\
& = f(p_m, s_i, \gamma_{p_m}) && (1) \\
& \leq f(q_l, s_j, \gamma_{p_m}) && (2) \\
& = g(s_0, s_j) + g(s_j, q_l) + h(q_l) + overshot(q_l, s_j, \gamma_{p_m}) \\
& \leq g(s_0, s_j) + g(s_j, q_l) + h^*(q_l) + overshot^*(q_l, s_j, \gamma_{p_m}) && (3) \\
& \leq g(s_0, s_j) + g(s_j, q_l) + h^*(q_l) + overshot^*(q_l, s_j, \gamma_{q_l}) && (4) \\
& \leq g(s_0, s_j) + g(s_j, q_n) + overshot^*(q_n, s_j, \gamma_{q_n}) && (5) \\
& = f^*(q_n, s_j, \gamma_{q_n})
\end{aligned}
$$

The true cost of the solution π_1 is $f^*(p_m, s_i, \gamma_{p_m}) = g(s_0, s_i) + g(s_i, p_m) + overshot^*(p_m, s_i, \gamma_{p_m})$. Following the search structure of SRE, at some point we chose to expand p_m. Since p_m is the last node on the path and our heuristic functions are admissible, the true cost f^* is equal to the cost function f (equality 1). Inequality 2 comes from our choice of the node p_m over some node q_l from σ_2. The admissibility of function η and the assumption of well-behaved recovery paths yields the admissibility of the function $overshot$. Having h and $overshot$ admissible with respect to h^* and $overshot^*$, we get inequality 3.

If the overshot would be calculated at some later point γ_{q_l} when exploring q_l, its value would be greater or equal to the value at time point γ_{p_m} (inequality 4). Finally, inequality 5 results from $\gamma_{q_n} + \eta^*(q_n) = \gamma_{q_l} + \eta^*(q_l)$ and the fact that $g(s_j, q_l) + h^*(q_l) \leq g(s_j, q_n)$. □

We initially described SRE as A* with multiple potential starting nodes that accounts for planning times. A different intuition can be obtained by thinking of SRE as similar to running multiple instances of A* with different initial states, and giving them computation time depending on the value of their f-function, combined with reasoning about the time that has passed and the time needed to finish the search. However, different search instances may influence each other, as the time passes for all of them simultaneously and thus affects their reasoning about planning time.

In SRE, the set of reference nodes R is considered to be supplied by the user (a parameter the user can adjust depending on the application). If R would have many elements (e.g., all the nodes of the original plan), it would saturate the processor, but the decision on when to deviate from the original plan would not be limited by sparsity of the set of reference nodes. If $R = \{s_n\}$, containing only the last node of the original plan, then SRE collapses to the baseline that always finishes execution before starting a new plan, thus missing out on the opportunities to deviate from the original plan to make progress towards the new job.

Planning for only one node limits the possibilities the agent has, but focuses the effort (there is no split attention between paths from different reference nodes). Thus, even though SRE offers the advantage of deviating from the original path sooner and finding a better path that way, there is no guarantee it will always outperform the baseline. Consider a scenario in which $R = \{r, s_n\}$ and the optimal path starts from s_n. Furthermore, the planning time, if planned only for s_n as the initial node, is exactly the time that the agent will take to execute the original path (so the baseline does not have any waiting time in s_n). If at any point, due to imprecise heuristic functions, SRE would explore a node with r as its reference node, the time would be irretrievably lost and the agent would have to wait in s_n until planning is finished.

The same is true for the second baseline we are considering: stopping the execution immediately when a new job arrives. If moving any further along the original plan is getting the agent further away from the new goal (and the old goal may also be achieved on a plan towards the new one), SRE will be outperformed as it will have to move back eventually.

Having noted the situations in which the baselines outperform SRE, outside the edge cases, SRE's parallel planning and execution on the one side, and the flexibility in choosing when to deviate from the original plan on the other side, makes it better suited for tasks with jobs arriving online.

Experiments

We implemented SRE in Fast Downward (Helmert 2006). In our implementation, we use a standard A* open list for each reference node, using the SRE extensions to the f-function only to select the open list to be used for the next expansion to avoid having to re-sort the open list. When overshooting a reference node, our implementation assumes that each action has an inverse action with the same cost.

Like Bugsy (Burns, Ruml, and Do 2013), we estimate the remaining number of expansions as $\eta(m) = \textit{delay} * d(m)$ (Dionne, Thayer, and Ruml 2011), where *delay* is the (moving) average number of expansions between inserting a node into the open list and expanding it, and d is an estimation of the remaining steps to the goal (like h, but ignoring action costs) under node m. The expansion delay is important to counteract *search vacillation* (Dionne, Thayer, and Ruml 2011), referring to the search fluctuating between different solution paths and, in our case, potentially of different reference nodes.

Our key performance metric is the *total time*, i.e. overall time for planning and execution. We are using an instance-specific factor to translate plan cost into execution time as a number of expansions, so the total time is also measured in number of expansions.

In all experiments, the popular FF heuristic (Hoffmann and Nebel 2001) is used to guide the search. For the expan-

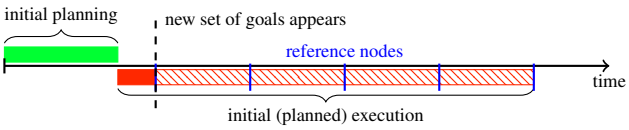
sion delay, we use a moving average over the last 100 expansions. The experiments were run on a cluster of Intel Xeon E5-2660 CPUs with a clock rate of 2.20 GHz. The time and memory limits were set to 30 minutes respectively 4 GB.

Benchmarks

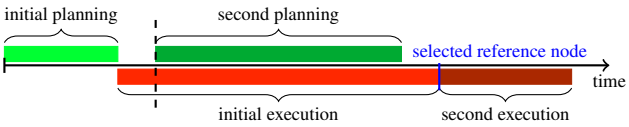
We adapted the IPC domains Elevators, Logistics, Rovers, Tidybot, Transport, and VisitAll to our setting, as representatives of applications where goals have a job-like nature in the sense of (i) recoverable states and (ii) stable goals. We included some variance though to test borderline situations. Elevators, Logistics, Transport, and VisitAll satisfy (i) and (ii), plus the additional assumption that an action sequence $\vec{\alpha}$ and its recovery sequence $\overleftarrow{\alpha}$ have the same cost. Rovers also satisfies (i) and (ii), but not the same-cost assumption: actions like taking an image don't need to be inverted. In assuming the opposite, our implementation is pessimistic which may adversely affect the plan cost reported. In Tidybot, finally, there are cases where objects are placed behind each other, and the robot cannot reach behind the object in the front. We added an "un-finish" action to ensure (i) recoverability. However, previously finished objects must be picked up again in these cases. Thus the nice-to-have condition (ii) is not satisfied.

The instances were adapted by splitting the set of goals in two: the first half is available in the beginning, and the other one becomes available later. The second set of goals is scheduled to appear during the execution of the first computed plan to obtain interesting instances. Since we are interested in a combination of planning and execution time, we need to convert both into the same unit.

A run of SRE on one such instance will look as follows:



The initially computed plan is being executed as a new job arrives. Here, the planner considers 5 reference nodes as potential initial states for the new plan.



The planner has computed an updated plan that starts from the second to last reference node. The initially computed plan is executed until that point before switching to the new plan. The total time is the time from the start of the first planning phase to the end of the overall execution.

In order to obtain interesting benchmark instances, we tried to ensure that the second planning phase starts and ends during the first planned execution. Thus, we let the second set of goals appear after a fraction of 0.1 of the initial plan is executed. We estimated the length of the second planning phase by running the planner offline with all goals enabled,

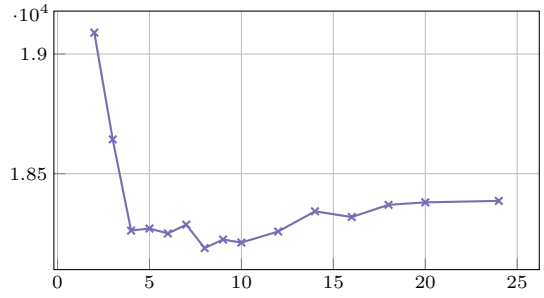


Figure 1: Total time as geometric mean over all instances (Y-axis) for SRE with different numbers of reference nodes (X-axis).

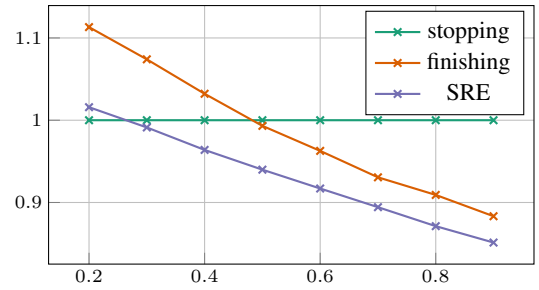


Figure 2: Total time as geometric mean over all instances relative to stopping and re-planning immediately (Y-axis) for $E = 0.2, 0.3, \ldots, 0.9$ (X-axis).

and used that to generate instances where the second planning phase is estimated to end at $E = 0.2, 0.3, \ldots, 0.9$ of the initially planned execution. This is achieved by adjusting the factor for the translation of the action cost to execution time, thereby changing the duration of the initially planned execution.

Results

SRE has one important parameter: the selection of the reference nodes. In our implementation, we set a number of reference nodes n_R, which are then selected in uniform intervals from the current plan. Figure 1 shows the total time (in number of expansions) for different values of n_R across our full benchmark set. If there are too few reference nodes, the algorithm does not have the best starting point for the next plan available. On the other hand, the performance also decreases slightly if too many reference nodes are used, as it becomes more difficult to settle on the most promising one quickly (especially if the planning time estimation is not very accurate). On average, SRE chooses nodes for expansion corresponding to the reference node which is used for the solution 34% of the time, more for fewer reference nodes (43% for $n_R = 3$), and less the more reference nodes are used (28% for $n_r = 24$). The overall best results are obtained with $n_R = 8$, and we use that setting for the remaining experiments.

We compare SRE to the two baselines: (a) finishing execution while planning only for the new goals and (b) stop-

ping execution and re-planning immediately. Figure 2 shows the results for different expected end points of the second planning phase, as total time relative to the performance of stopping and re-planning immediately. If the planning time is very short compared to the execution time (small values of E), stopping works well. However, if planning is non-trivial ($E \geq 0.3$), SRE performs better. Furthermore, SRE always outperforms baseline (a), for all values of E on all domains. On average, SRE reduces the total time by 6.9% compared to stopping and re-planning immediately, and by 5.6% compared to finishing the planned execution. The results are similar across all domains, except that the relative strength of the baselines differs. On Transport and VisitAll, stopping is better than finishing for $E \leq 0.6$ respectively $E \leq 0.7$, though SRE is the best algorithm for $E \geq 0.4$. On Rovers, stopping is only better than finishing for $E = 0.2$, and SRE is the best algorithm for all values of E. The biggest advantage over both baselines is obtained in Elevators, with a total time reduction 7.1% and 7% over stopping and finishing respectively.

Both baselines waste time, though in different ways. Halting the execution is inefficient as the agent is idle while planning. Finishing the execution exploits the parallelism of proceeding with the execution. However, planning only for the second set of goals is usually quite fast, and there would be more time available while waiting for the execution of the initial plan to finish. SRE uses this time more efficiently to compute better overall plans, and effectively improves the combined planning and execution time over both baselines.

Conclusion

Many planning applications feature the arrival of new jobs while a plan is already being executed. We introduced an algorithm, SRE, which solves this problem effectively: planning simultaneously for multiple potential initial states while proceeding with the execution. The algorithm is aware of its own planning time to select such an initial state in an informed manner. On a set of planning benchmarks, SRE clearly outperforms both vanilla solutions, (a) finishing execution prior to executing the new plan, and (b) stopping execution and waiting for re-planning to terminate.

An interesting question for future research is whether our approach can be extended to, and be useful in, situations with unrecoverable states, i.e., where goals may be in direct conflict. We also believe that our ideas may be brought to bear on domain-specific solutions to achieve better performance, for example in warehouse logistics.

Acknowledgements

This research was sponsored in part by the ERC Synergy project 610150 (ImPACT) and by the German Research Foundation (DFG) under grants HO 2169/5-1, "Critically Constrained Planning via Partial Delete Relaxation", and 389792660 as part of TRR 248 (see https://perspicuous-computing.science).

References

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

Burns, E.; Benton, J.; Ruml, W.; Yoon, S. W.; and Do, M. B. 2012. Anticipatory on-line planning. In McCluskey, L.; Williams, B. C.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*. AAAI.

Burns, E.; Ruml, W.; and Do, M. B. 2013. Heuristic search when time matters. *J. Artif. Intell. Res.* 47:697–740.

Cashmore, M.; Coles, A.; Cserna, B.; Karpas, E.; Magazzeni, D.; and Ruml, W. 2018. Temporal planning while the clock ticks. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018.*, 39–46.

Daum, J.; Torralba, Á.; Hoffmann, J.; Haslum, P.; and Weber, I. 2016. Practical undoability checking via contingent planning. In Coles, A. J.; Coles, A.; Edelkamp, S.; Magazzeni, D.; and Sanner, S., eds., *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016.*, 106–114. AAAI Press.

Dionne, A. J.; Thayer, J. T.; and Ruml, W. 2011. Deadline-aware search using on-line measures of behavior. In Borrajo, D.; Likhachev, M.; and Linares López, C., eds., *Proceedings of the Fourth Annual Symposium on Combinatorial Search, SOCS 2011, Castell de Cardona, Barcelona, Spain, July 15.16, 2011*. AAAI Press.

Estlin, T.; Rabideau, G.; Mutz, D.; and Chien, S. 2000. Using continuous planning techniques to coordinate multiple rovers. *Electronic Transactions on Artificial Intelligence* 4:45–57.

Fox, M., and Long, D. 2003. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.* 20:61–124.

Fox, M.; Gerevini, A.; Long, D.; and Serina, I. 2006. Plan stability: Replanning versus plan repair. In Long, D.; Smith, S. F.; Borrajo, D.; and McCluskey, L., eds., *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling, ICAPS 2006, Cumbria, UK, June 6-10, 2006*, 212–221. AAAI.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2004. *Automated planning - theory and practice*. Elsevier.

Ghallab, M.; Nau, D. S.; and Traverso, P. 2016. *Automated Planning and Acting*. Cambridge University Press.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Systems Science and Cybernetics* 4(2):100–107.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Intell. Res.* 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.

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

Knight, R.; Rabideau, G.; Chien, S. A.; Engelhardt, B.; and Sherwood, R. 2001. Casper: Space exploration through continuous planning. *IEEE Intelligent Systems* 16(5):70–75.

Lemons, S.; Benton, J.; Ruml, W.; Do, M. B.; and Yoon, S. W. 2010. Continual on-line planning as decision-theoretic incremental heuristic search. In *Embedded Reasoning, Papers from the 2010 AAAI Spring Symposium, Technical Report SS-10-04, Stanford, California, USA, March 22-24, 2010*. AAAI.

Myers, K. L. 1999. CPEF: A continuous planning and execution framework. *AI Magazine* 20(4):63–69.

Ruml, W.; Do, M. B.; Zhou, R.; and Fromherz, M. P. J. 2011. On-line planning and scheduling: An application to controlling modular printers. *J. Artif. Intell. Res.* 40:415–468.

Ruml, W.; Do, M. B.; and Fromherz, M. P. J. 2005. On-line planning and scheduling for high-speed manufacturing. In Biundo, S.; Myers, K. L.; and Rajan, K., eds., *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*, 30–39. AAAI.

Guiding MCTS with Generalized Policies for Probabilistic Planning

William Shen[1], Felipe Trevizan[1], Sam Toyer[2], Sylvie Thiébaux[1] and **Lexing Xie[1]**
[1] The Australian National University [2] University of California, Berkeley
[1]firstname.lastname@anu.edu.au [2]sdt@berkeley.edu

Abstract

We examine techniques for combining generalized policies with search algorithms to exploit the strengths and overcome the weaknesses of each when solving probabilistic planning problems. The Action Schema Network (ASNet) is a recent contribution to planning that uses deep learning and neural networks to learn generalized policies for probabilistic planning problems. ASNets are well suited to problems where local knowledge of the environment can be exploited to improve performance, but may fail to generalize to problems they were not trained on. Monte-Carlo Tree Search (MCTS) is a forward-chaining state space search algorithm for optimal decision making which performs simulations to incrementally build a search tree and estimate the values of each state. Although MCTS can achieve state-of-the-art results when paired with domain-specific knowledge, without this knowledge, MCTS requires a large number of simulations in order to obtain reliable estimates in the search tree. By combining ASNets with MCTS, we are able to improve the capability of an ASNet to generalize beyond the distribution of problems it was trained on, as well as enhance the navigation of the search space by MCTS.

1 Introduction

Planning is the essential ability of a rational agent to solve the problem of choosing which actions to take in an environment to achieve a certain goal. This paper is mainly concerned with combining the advantages of forward-chaining state space search through UCT (Kocsis and Szepesvári 2006), an instance of Monte-Carlo Tree Search (MCTS) (Browne et al. 2012), with the domain-specific knowledge learned by Action Schema Networks (ASNets) (Toyer et al. 2018), a domain-independent learning algorithm. By combining UCT and ASNets, we hope to more effectively solve planning problems, and achieve the best of both worlds.

The Action Schema Network (ASNet) is a recent contribution in planning that uses deep learning and neural networks to learn generalized policies for planning problems. A generalized policy is a policy that can be applied to any problem from a given planning domain. Ideally, this generalized policy is able to reliably solve all problems in the given domain, although this is not always feasible. ASNets are well suited to problems where "local knowledge of the environment can help to avoid certain traps" (Toyer et al. 2018). In such problems, an ASNet can significantly outperform traditional planners that use heuristic search. Moreover, a significant advantage of ASNets is that a network can be trained on a limited number of small problems, and generalize to problems of any size. However, an ASNet is not guaranteed to reliably solve all problems of a given domain. For example, an ASNet could fail to generalize to difficult problems that it was not trained on – an issue often encountered with machine learning algorithms. Moreover, the policy learned by an ASNet could be suboptimal due to a poor choice of hyperparameters that has led to an undertrained or overtrained network. Although our discussion is closely tied to ASNets, our contributions are more generally applicable to any method of learning a (generalized) policy.

Monte-Carlo Tree Search (MCTS) is a state-space search algorithm for optimal decision making which relies on performing Monte-Carlo simulations to build a search tree and estimate the values of each state (Browne et al. 2012). As we perform more and more of these simulations, the state estimates become more accurate. MCTS-based game-playing algorithms have often achieved state-of-the-art performance when paired with domain-specific knowledge, the most notable being AlphaGo (Silver et al. 2016). One significant limitation of vanilla MCTS is that we may require a large number of simulations in order to obtain reliable estimates in the search tree. Moreover, because simulations are random, the search may not be able to sense that certain branches of the tree will lead to sub-optimal outcomes. We are concerned with UCT, a variant of MCTS that balances the trade-off between exploration and exploitation. However, our work can be more generally used with other search algorithms.

Combining ASNets with UCT achieves three goals. (1) *Learn what we have not learned:* improve the capability of an ASNet to generalize beyond the distribution of problems it was trained on, and of UCT to bias the exploration of actions to those that an ASNet wishes to exploit. (2) *Improve on sub-optimal learning:* obtain reasonable evaluation-time performance even when an ASNet was trained with sub-optimal hyperparameters, and allow UCT to converge to the optimal action in a smaller number of *trials*. (3) *Be robust to changes in the environment or domain:* improve perfor-

This paper is subsumed by "Guiding Search with Generalized Policies for Probabilistic Planning", which has been published in the Symposium on Combinatorial Search 2019.

mance when the test environment differs substantially from the training environment.

The rest of the paper is organized as follows. Section 2 formalizes probabilistic planning as solving a Stochastic Shortest Path problem and gives an overview of ASNets and MCTS along with its variants. Section 3 defines a framework for *Dynamic Programming UCT (DP-UCT)* (Keller and Helmert 2013). Next, Section 4 examines techniques for combining the policy learned by an ASNet with *DP-UCT*. Section 5 then presents and analyzes our results. Finally, Section 6 summarizes our contributions and discusses related and future work.

2 Background

A Stochastic Shortest Path problem (SSP) is a tuple $\langle S, s_0, G, A, P, C\rangle$ (Bertsekas and Tsitsiklis 1991) where S is the finite set of states, $s_0 \in S$ is the initial state, $G \subseteq S$ is the finite set of goal states, A is the finite set of actions, $P(s' \mid a, s)$ is the probability that we transition into s' after applying action a in state s, and $C(s, a) \in (0, \infty)$ is the cost of applying action a in state s. A solution to an SSP is a stochastic policy $\pi\colon A \times S \to [0, 1]$, where $\pi(a \mid s)$ represents the probability action a is applied in the current state s. An optimal policy π^*, is a policy that selects actions which minimize the expected cost of reaching a goal. For SSPs, there always exists an optimal policy that is deterministic which may be obtained by finding the fixed-point of the state-value function V^* known as the Bellman optimality equation (Bertsekas and Tsitsiklis 1991), and the action-value function Q^*. That is, in the state s, we obtain π^* by finding the action a that minimizes $Q^*(s, a)$.

$$V^*(s) = \begin{cases} 0 & \text{if } s \in G \\ \min_{a \in A} Q^*(s, a) & \text{otherwise} \end{cases}$$

$$Q^*(s, a) = C(s, a) + \sum_{s' \in S} P(s' \mid a, s) \cdot V^*(s')$$

We handle dead ends using the finite-penalty approach (Kolobov, Mausam, and Weld 2012). That is, we introduce a fixed dead-end penalty $D \in (0, \infty)$ which acts as a limit to bound the maximum expected cost to reach a goal, and a *give-up* action which is selected if the expected cost is greater than or equal to D.

2.1 Action Schema Networks (ASNets)

The ASNet is a neural network architecture that exploits deep learning techniques in order to learn generalized policies for probabilistic planning problems (Toyer et al. 2018). An ASNet consists of alternating action layers and proposition layers (Figure 1), where the first and last layer are always action layers. The output of the final layer is a stochastic policy $\pi\colon A \times S \to [0, 1]$.

An action layer is composed of a single action module for each ground action in the planning problem. Similarly, a proposition layer is composed of a single proposition module for each ground proposition in the problem. These modules are sparsely connected, ensuring that only the relevant action modules in one layer are connected to a proposition module in the next layer. An action module in one layer is connected to a proposition module in the next layer only if the ground proposition appears in the preconditions or effects of a ground action. Similarly, a proposition module in one layer is connected to an action module in the next layer only if the ground proposition appears in the preconditions or effects of the relevant ground action. Since all ground actions instantiated from the same action schema will have the same structure, we can share the same set of weights between their corresponding action modules in a single action layer. Similarly, weights are shared between proposition modules in a single proposition layer that correspond to the same predicate. It is easy to see that by learning a set of common weights θ for each action schema and predicate, we can scale an ASNet to any problem of the same domain.

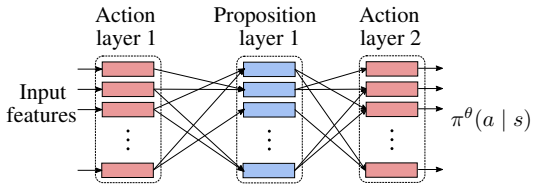


Figure 1: ASNet with 1 hidden layer (Toyer et al. 2018)

ASNets only have a fixed number of layers, and are thus unable to solve all problems in domains that require arbitrarily long chains of reasoning about action–proposition relationships. Moreover, like most machine learning algorithms, an ASNet could fail to generalize to new problems if not trained properly. This could be due to a poor choice of hyperparameters, overfitting to the problems the network was trained on, or an unrepresentative training set.

2.2 Monte-Carlo Tree Search (MCTS)

MCTS is a state-space search algorithm that builds a search tree in an incremental manner by performing *trials* until we reach some computational budget (e.g. time, memory) at each decision step (Browne et al. 2012), at which point MCTS returns the action that gives the best estimated value.

A trial is composed of four phases. Firstly, in the selection phase, MCTS recursively selects nodes in the tree using a child selection policy until it encounters an unexpanded node, i.e. a node without any children. Next, in the expansion phase, one or more child nodes of the leaf node are created in the search tree according to the available actions. Now, in the simulation phase, a simulation of the scenario is played-out from one of the new child nodes until we reach a goal or dead end, or exceed the computational budget. Finally, in the backpropagation phase, the result of this trial is backpropagated through the selected nodes in the tree to update their estimated values. The updated estimates affect the child selection policy in future trials.

Upper Confidence Bounds applied to Trees (UCT) (Kocsis and Szepesvári 2006) is a variant of MCTS that addresses the trade-off between the exploration of nodes that have not been visited often, and the exploitation of nodes that currently have good state estimates. UCT treats

the choice of a child node as a multi-armed bandit problem by selecting the node which maximizes the Upper Confidence Bound 1 (UCB1) term, which we detail in the selection phase in Section 3.1.

Trial-Based Heuristic Tree Search (THTS) (Keller and Helmert 2013) is an algorithmic framework that generalizes MCTS, dynamic programming, and heuristic search planning algorithms. In a THTS algorithm, we must specify five ingredients: action selection, backup function, heuristic function, outcome selection and the trial length. We discuss these ingredients and a modified version of THTS to additionally support UCT with ASNets in Section 3.

Using these ingredients, Keller and Helmert (2013) create three new algorithms, all of which provide superior theoretical properties over UCT: MaxUCT, Dynamic Programming UCT (DP-UCT) and UCT*. DP-UCT and its variant UCT*, which use *Bellman backups*, were found to outperform original UCT and MaxUCT. Because of this, we will focus on DP-UCT, which we formally define in the next section.

3 DP-UCT Framework

Our framework is a modification of DP-UCT from THTS. It is designed for SSPs with dead ends instead of finite horizon MDPs and is focused on minimizing the cost to a goal rather than maximizing rewards. It also introduces the *simulation function*, a generalization of random rollouts used in MCTS.

We adopt the representation of alternating decision nodes and chance nodes in our search tree, as seen in THTS. A decision node n_d is a tuple $\langle s, C^k, V^k, \{n_1, \ldots, n_m\}\rangle$, where $s \in S$ is the state, $C^k \in \mathbb{Z}_0^+$ is the number of visits to the node in the first k trials, $V^k \in \mathbb{R}_0^+$ is the state-value estimate based on the first k trials, and $\{n_1, \ldots, n_m\}$ are the successor nodes (i.e. children) of n_d. A chance node n_c is a tuple $\langle s, a, C^k, Q^k, \{n_1, \ldots, n_m\}\rangle$, where additionally, $a \in A$ is the action, and Q^k is the action-value estimate based on the first k trials.

We use $V^k(n_d)$ to refer to the state-value estimate of a decision node n_d, $a(n_c)$ to refer to the action of a chance node n_c, and so on for all the elements of n_d and n_c. Additionally, we use $S(n)$ to represent the successor nodes $\{n_1, \ldots, n_m\}$ of a search node n, and we also employ the shorthand $P(n_d \,|\, n_c) = P(s(n_d) \,|\, a(n_c), s(n_c))$ and $c(n_c) = c(s(n_c), a(n_c))$. Initially, the search tree contains a single decision node n_d with $s(n_d) = s_0$, representing the initial state of our problem.

3.1 Algorithm

UCT is described as an online planning algorithm, as it interleaves planning with execution. At each decision step, UCT returns an action either when a time cutoff is reached, or a maximum number of trials is performed. UCT then selects the chance node n_c from the children of the root decision node that has the highest action-value estimate, $Q^k(n_c)$, and applies its action $a(n_c)$. We sample a decision node n_d from $S(n_c)$ based on the transition probabilities $P(n_d \,|\, n_c)$ and set n_d to be the new root of the tree.

A single trial under our framework consists of the selection, expansion, simulation and backup phase.

Selection Phase. As described in THTS, in this phase we traverse the explicit nodes in the search tree by alternating between *action selection* for decision nodes, and *outcome selection* for chance nodes until we reach an unexpanded decision node n_d, which we call the tip node of the trial.

Action selection is concerned with selecting a child chance node n_c from the successors $S(n_d)$ of a decision node n_d. UCT selects the child chance node that maximizes the UCB1 term, i.e. $\arg\max_{n_c \in S(n_d)} \mathrm{UCB1}(n_d, n_c)$, where

$$\mathrm{UCB1}(n_d, n_c) = \underbrace{B \cdot \sqrt{\frac{\log C^k(n_d)}{C^k(n_c)}}}_{\text{exploration}} - \underbrace{Q^k(n_c)}_{\text{exploitation}} .$$

B is the bias term which allows us to adjust the trade-off between exploration and exploitation. We set $\mathrm{UCB1}(n_d, n_c) = \infty$ if $C^k(n_c) = 0$ to force the exploration of chance nodes that have not been visited.

In outcome selection, we randomly sample an outcome of an action, i.e. sample a child decision node n_d from the successors $S(n_c)$ of a chance node n_c based on the transition probabilities $P(n_d \,|\, n_c)$.

Expansion Phase. In this phase, we expand the tip node n_d and optionally initialize the Q-values of its child chance nodes, $S(n_d)$. Calculating an estimated Q-value requires calculating a weighted sum of the form:

$$Q^k(n_c) = c(n_c) + \sum_{n_d \in S(n_c)} P(n_d \,|\, n_c) \cdot H(s(n_d)) \,,$$

where H is some domain-independent SSP heuristic function such as h^{add}, $h^{\max}$, h^{pom}, or h^{roc} (Teichteil-Königsbuch, Vidal, and Infantes 2011; Trevizan, Thiébaux, and Haslum 2017). This can be expensive when n_c has many successor decision nodes.

Simulation Phase. Immediately after the expansion phase, we transition to the simulation phase. Here we perform a simulation (also known as a rollout) of the planning problem from the tip node's state $s(n_d)$, until we reach a goal or dead-end state, or exceed the *trial length*. This stands in contrast to the behaviour of THTS, which lacks a simulation phase and would continuously switch between the selection and expansion phases until the *trial length* is reached.

We use the *simulation function* to choose which action to take in a given state, and sample the next state according to the transition probabilities. If we complete a simulation without reaching a goal or dead end, we add a heuristic estimate $H(s')$ to the rollout cost, where s' is the final rollout state. If s' is a dead end, then we set the rollout cost to be the dead-end penalty D.

The trial length bounds how many steps can be applied in the simulation phase, and hence allows us to adjust the lookahead capability of DP-UCT. By setting the trial length to be very small, we can focus the search on nodes closer to the root of the tree, much like breadth-first search (Keller and Helmert 2013). Following the steps above, if the trial length is 0, we do not perform any simulations and simply take a heuristic estimate for the tip node of the trial, or D if the tip node represents a dead-end.

Traditional MCTS-based algorithms use a random simulation function, where each available action in the state has the same probability of being selected. However, this is not very suitable for SSPs as we can continuously loop around a set of states and never reach a goal state. Moreover, using a random simulation function requires an extremely large number of simulations to obtain good estimates for state-values and action-values within the search tree. Because of this, the simulation phase in MCTS-based algorithms for planning is often neglected and replaced by a heuristic estimate. This is equivalent to setting the trial length to be 0, where we backup a heuristic estimate once we expand the tip node of the trial.

However, there can be situations where the heuristic function is misleading or uninformative and thus misguides the search. In such a scenario, it could be more productive to use a random simulation function, or a simulation function influenced by domain-specific knowledge (i.e., the knowledge learned by an ASNet) to calculate estimates.

Backup Phase. After the simulation phase, we must propagate the information we have gained from the current trial back up the search tree. We use the *backup function* to update the state-value estimate $V^k(n_d)$ for decision nodes and the action-value estimate $Q^k(n_c)$ for chance nodes. We do this by propagating the information we gained during the simulation in reverse order through the nodes in the trial path, by continuously applying the backup function for each node until we reach the root node of the search tree.

Original UCT is defined with Monte-Carlo backups, in which the transition model is unknown and hence estimated based on the number of visits to nodes. However, in our work we consider the transition model to be known a priori. For that reason, DP-UCT only considers Bellman backups (Keller and Helmert 2013), which additionally take the probabilities of outcomes into consideration when backing up action value estimates $Q^k(n_c)$:

$$V^k(n_d) = \begin{cases} 0 & \text{if } s(n_d) \text{ is a goal} \\ D & \text{if } s(n_d) \text{ is a dead end} \\ \min\limits_{n_c \in S(n_d)} Q^k(n_c) & \text{otherwise,} \end{cases}$$

$$Q^k(n_c) = \min \left\{ D, c(n_c) + \sum_{n_d \in \Upsilon^k(n_c)} \hat{P}(n_d \,|\, n_c) \cdot V^k(n_d) \right\},$$

$$\text{where } \Upsilon^k(n_c) = \left\{ n_d \,|\, n_d \in S(n_c),\ C^k(n_d) > 0 \right\},$$

$$\text{and } \hat{P}(n_d \,|\, n_c) = \frac{P(n_d \,|\, n_c)}{\sum_{n'_d \in \Upsilon^k(n_c)} P(n'_d \,|\, n_c)}.$$

$\Upsilon^k(n_c)$ represents the child decision nodes of n_c that have already been visited in the first k trials and hence have state-value estimates. Thus, $\hat{P}(n_d \,|\, n_c)$ allows us to weigh the state-value estimate $V^k(n_d)$ of each visited child decision node n_d proportionally by its probability $P(n_d \,|\, n_c)$ and that of the unvisited child decision nodes.

It should be obvious that Bellman backups are derived directly from the Bellman optimality equations we presented in Section 2. Thus a flavor of UCT using Bellman backups is asymptotically optimal given a correct selection of ingredients that will ensure all nodes are explored infinitely often.

4 Combining DP-UCT with ASNets

4.1 Using ASNets as a Simulation Function

Recall that an ASNet learns a stochastic policy $\pi\colon A \times S \to [0, 1]$, where $\pi(a \,|\, s)$ represents the probability action a is applied in state s. We introduce two simulation functions which make use of a trained ASNet: STOCHASTIC ASNETS which simply samples from the probability distribution given by π to select an action, and MAXIMUM ASNETS which selects the action with the highest probability – i.e. $\arg\max_{a \in A(s)} \pi(a \,|\, s)$.

Since the navigation of the search space is heavily influenced by the state-value and action-value estimates we obtain from performing simulations, DP-UCT with an ASNet-based simulation function would ideally converge to the optimal policy in a smaller number of simulations compared to if we used the random simulation function. Of course, we expect this to be the case if an ASNet has learned some useful features or tricks about the environment or domain of the problem we are tackling.

However, using ASNets as a simulation function may not be very robust if the learned policy is misleading and uninformative. Here, robustness indicates how well UCT can recover from the misleading information it has been provided. In this situation, DP-UCT with ASNets as a simulation function would require a significantly larger number of simulations in order to converge to the optimal policy than DP-UCT with a random simulation function. Regardless the quality of the learned policy, DP-UCT remains asymptotically optimal when using an ASNet-based simulation function if the selection of ingredients guarantees that our search algorithm will explore all nodes infinitely often. Nonetheless, an ASNet-based simulation function should only be used if its simulation from the tip node n_d better approximates $V^*(n_d)$ than a heuristic estimate $H(s(n_d))$.

Choosing between STOCHASTIC ASNETS and MAXIMUM ASNETS. We can perceive the probability distribution given by the policy π of an ASNet to represent the 'confidence' the network has in applying each action. Obviously, MAXIMUM ASNETS will completely bias the simulations towards what an ASNet believes is the best action for a given state. If the probability distribution is highly skewed towards a single action, then MAXIMUM ASNETS would be the better choice, as the ASNet is very 'confident' in its decision to choose the corresponding action. On the other hand, if the probability distribution is relatively uniform, then STOCHASTIC ASNETS would likely be the better choice. In this situation, the ASNet may be uncertain and not very 'confident' in its decision to choose among a set of actions. Thus, to determine which ASNet-based simulation function to use, we should carefully consider to what extent an ASNet is able to solve the given problem reliably.

4.2 Using ASNets in UCB1

The UCB1 term allows us to balance the trade-off between exploration of actions in the search tree that have not been

applied often, and exploitation of actions that we already know have good action-value estimates based on previous trials. By including an ASNet's influence within UCB1 through its policy π, we hope to maintain this fundamental trade-off yet further bias the action selection to what the ASNet believes are promising actions.

Simple ASNet Action Selection. We select the child chance node n_c of a decision node n_d that maximizes:

$$\text{SIMPLE-ASNET}(n_d, n_c) = \frac{M \cdot \pi(n_c)}{C^k(n_c)} + \text{UCB1}(n_d, n_c)$$

$$= \underbrace{\frac{M \cdot \pi(n_c)}{C^k(n_c)} + B \cdot \sqrt{\frac{\log C^k(n_d)}{C^k(n_c)}}}_{exploration} - \underbrace{Q^k(n_c)}_{exploitation}$$

where $M \in \mathbb{R}^+$ and $\pi(n_c) = \pi(a(n_c) \mid s(n_c))$ for the stochastic policy π learned by ASNet. Similar to UCB1, if a child chance node n_c has not been visited before (i.e., $C^k(n_c) = 0$), we set $\text{SIMPLE-ASNET}(n_d, n_c) = \infty$ to force its exploration. The new parameter M, called the influence constant, allows us to control the exploitation of an ASNet's policy π for exploration and, the higher M is, the higher the influence of the ASNet in the action selection.

Notice that the influence of the ASNet diminishes as we apply the action $a(n_c)$ more often because $M \cdot \pi(n_c)/C^k(n_c)$ decreases as the number of visits to the chance node n_c increases. Moreover, since the bias provided by $M \cdot \pi(n_c)/C^k(n_c)$ diminishes to 0 as $C^k(n_c) \to \infty$ faster than $B \cdot \sqrt{\log C^k(n_d)/C^k(n_c)}$ (i.e., the original UCB1 bias term), SIMPLE-ASNET preserves the asymptotic optimality of UCB1: as $C^k(n_c) \to \infty$, $\text{SIMPLE-ASNET}(n_d, n_c)$ equals $\text{UCB1}(n_d, n_c)$ and both converge to the optimal action-value $Q^*(n_c)$ (Kocsis and Szepesvári 2006).

Because of this similarity with UCB1 and their same initial condition (i.e., treating divisions by $C^k(n_c) = 0$ as ∞), we expect that SIMPLE-ASNET action selection will be robust to any misleading information provided by the policy of a trained ASNet. Nonetheless, the higher the value of the influence constant M, the more trials we require to combat any uninformative information.

Ranked ASNet Action Selection. One pitfall of the infinite exploration bonus in SIMPLE-ASNET action selection when $C^k(n_c) = 0$ is that all child chance nodes must be visited at least once before we actually exploit the policy learned by an ASNet. Ideally, we should use the knowledge learned by an ASNet to select the order in which unvisited chance nodes are explored. Thus, we introduce RANKED-ASNET action selection, an extension to SIMPLE-ASNET action selection.

$$\text{RANKED-ASNET}(n_d, n_c) =$$

$$\begin{cases} \text{SIMPLE-ASNET}(n_d, n_c) & \text{if } \forall n'_c \in S(n_d), C^k(n'_c) > 0 \\ \pi(n_c) & \text{if } C^k(n_c) = 0 \\ -\infty & \text{otherwise} \end{cases}$$

The first condition stipulates that all chance nodes are selected and visited at least once before SIMPLE-ASNET action selection is used. Otherwise, chance nodes that have already been visited are given a value of $-\infty$, while the values of unvisited nodes correspond to their probability in the policy π. Thus, unvisited child chance nodes are visited in decreasing order of their probability within the policy π.

RANKED-ASNET action selection will allow DP-UCT to focus the initial stages of its search on what an ASNet believes are the most promising parts of the state space. Given that the ASNet has learned some useful knowledge of which action to apply at each step, we expect RANKED-ASNET action selection to require a smaller number of trials to converge to the optimal action in comparison with SIMPLE-ASNET action selection. However, RANKED-ASNET may not be as robust as SIMPLE-ASNET when the policy learned by an ASNet is misleading or uninformative. For example, if the optimal action has the lowest probability among all actions in the ASNet policy and is hence explored last, then we would require an increased number of trials to converge to this optimum.

Comparison with ASNet-based Simulation Functions. DP-UCT with ASNet-influenced action selection is more robust to misleading information than DP-UCT with an ASNet-based simulation function. Since SIMPLE-ASNET and RANKED-ASNET action selection decreases the influence of a network as we apply an action it has suggested more frequently, we will eventually explore actions that may have a small probability in the policy learned by the ASNet but are in fact optimal. We would require a much larger number of trials to achieve this when using an ASNet-based simulation function, as the state-value and action-value estimates in the search tree would be directly derived from ASNet-based simulations.

5 Empirical Evaluation

5.1 Experimental Setup

All experiments were performed on an Amazon Web Services EC2 c5.4x large instance with 16 CPUs and 32GB of memory. Each experiment was limited to one CPU core with a maximum turbo clock speed of 3.5 GHz. No restrictions were placed on the amount of memory an experiment used.

Considered Planners. For our experiments, we consider two baseline planners: the original ASNets algorithm and *UCT**. The latter is a variation of DP-UCT where the trial length is 0 while still using UCB1 to select actions, Bellman backups as the backup function, and no simulation function. UCT* was chosen as a baseline because it outperforms original DP-UCT due to its stronger theoretical properties (Keller and Helmert 2013). We consider four parametrizations of our algorithms – namely, (i) Simple ASNets, (ii) Ranked ASNets, (iii) Stochastic ASNets, and (iv) Maximum ASNets – where: parametrizations (i) and (ii) are UCT* using SIMPLE and RANKED-ASNET action selection, respectively; and parametrizations (iii) and (iv) are DP-UCT with a problem-dependent trial length using STOCHASTIC and MAXIMUM ASNETS as the simulation function, respectively.

ASNet Configuration. We use the same ASNet hyperparameters as described by Toyer et al. to train each network. Unless otherwise specified, we imposed a strict two hour time limit to train the network, though in most situations, the network finished training within one hour. All ASNets were trained using an LRTDP-based (Bonet and Geffner 2003) teacher that used LM-cut (Helmert and Domshlak 2009) as the heuristic to compute optimal policies. We only report the time taken to solve each problem for the final results for an ASNet, and hence do not include the training time.

DP-UCT Configuration. For all DP-UCT configurations we used h^{add} (Bonet and Geffner 2001) as the heuristic function because it allowed DP-UCT to converge to a good solution in a reasonable time in our experiments, and set the UCB1 bias parameter B to $\sqrt{2}$. For all problems with dead ends, we enabled Q-value initialization, as it helps us avoid selecting a chance node for exploration that may lead to a dead end. We did not enable this for problems without dead ends because estimating Q-values is computationally expensive, and not beneficial in comparison to the number of trials that could have been performed in the same time frame.

We gave all configurations a 10 second time cutoff to do trials and limited the maximum number of trials to 10,000 at each decision step to ensure fairness. Moreover, we set the dead-end penalty to be 500. We gave each planning round a maximum time of 1 hour, and a maximum of 100 execution steps. We ran 30 rounds per planner for each experiment.

5.2 Domains

Stack Blocksworld. Stack Blocksworld is a special case of the deterministic Blocksworld domain in which the goal is to stack n blocks initially on the table into a single tower. We train an ASNet to unstack n blocks from a single tower and put them all down on the table. Since the network has never learned how to stack blocks, it completely fails at stacking the n blocks on the table into a single tower. A setting like this one—where the distributions of training and testing problems have non-overlapping support—represents a near-worst-case scenario for inductive learners like ASNets. In contrast, stacking blocks into a single tower is a relatively easy problem for UCT*. Our aim in this experiment is to show that DP-UCT can overcome the misleading information learned by ASNet policy. We train an ASNet on unstack problems with 2 to 10 blocks, and evaluate DP-UCT and ASNets on stack problems with 5 to 20 blocks.

Exploding Blocksworld. This domain is an extension of deterministic Blocksworld, and is featured in the International Probabilistic Planning Competitions (IPPC). In Exploding Blocksworld, putting down a block can detonate and destroy the block or the table it was put down on. Once a block or the table is exploded, we can no longer use it; therefore, this domain contains unavoidable dead ends. A good policy avoids placing a block down on the table or down on another block that is required for the goal state (if possible). It is very difficult for an ASNet to reliably solve Exploding Blocksworld problems as each problem could have its own 'trick' in order to avoid dead ends and reach the goal with minimal cost.

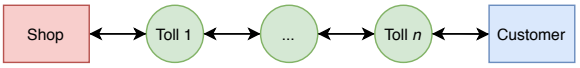


Figure 2: The CosaNostra Pizza Domain

We train an ASNet for 5 hours on a selected set of 16 problems (including those with avoidable and unavoidable dead ends) that were optimally solved by LRTDP within 2 minutes.[1] We evaluate ASNets and DP-UCT on the first eight problems from IPPC 2008 (Bryce and Buffet 2008). By combining DP-UCT and ASNets, we hope to exploit the limited knowledge and 'tricks' learned by an ASNet on the problems it was trained on to navigate the search space. That is, we aim to learn what we have not learned, and improve suboptimal learning.

CosaNostra Pizza (Toyer et al. 2018). The objective in CosaNostra Pizza is to safely deliver a pizza from the pizza shop to the waiting customer and then return to the shop. There is a series of toll booths on the two-way road between the pizza shop and the customer (Figure 2). At each toll booth, you can choose to either pay the toll operator or drive straight through without paying. We save a time step by driving straight through without paying but the operator becomes angry. Angry operators drop their toll gate on you and crush your car (leading to a dead end) with a probability of 50% when you next pass through their booth. Hence, the optimal policy is to only pay the toll operators on the trip to the customer, but not on the trip back to the pizza shop (as we will not revisit the booth). This ensures a safe return, as there will be no chance of a toll operator crushing your car at any stage. Thus, CosaNostra Pizza is an example of a problem with avoidable dead ends.

An ASNet is able to learn the trick of paying the toll operators only on the trip to the customer, and scales up to large instances while heuristic search planners based on determinisation (either for search or for heuristic computation) do not scale up (Toyer et al. 2018). The reason for the underperformance of determinisation-based techniques (e.g., using h^{add} as heuristic) is the presence of avoidable dead ends in the CosaNostra domain. Moreover, heuristics based on delete relaxation (e.g., h^{add}) also underperform in the CosaNostra domain because they consider that the agent crosses each toll booth only once, i.e., this relaxation ignores the return path since it uses the same propositions as the path to the customer. Thus, we expect UCT* to not scale up to larger instances since it will require extremely long reasoning chains in order to always pay the toll operator on the trip to the customer; however, by combining DP-UCT with the optimal policy learned by an ASNet, we expect to scale up to much larger instances than UCT* alone.

For the CosaNostra Pizza problems, we train an ASNet on problems with 1 to 5 toll-booths, and evaluate DP-UCT and ASNets on problems with 2 to 15 toll booths.

[1] The training problems are available here: https://s3.amazonaws.com/ex-blocksworld/problems.zip

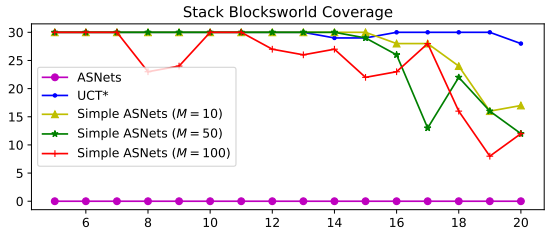


Figure 3: Coverage results for Stack Blocksworld.

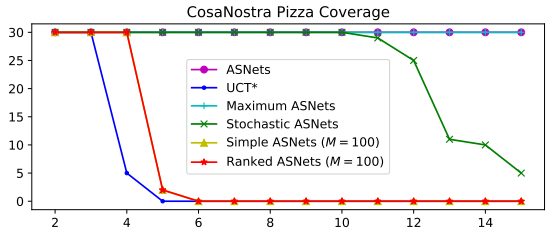


Figure 4: Coverage results for CosaNostra Pizza. Both ASNets and Maximum ASNets obtained perfect coverage.

5.3 Results

Stack Blocksworld. We allocate to each execution step $n/2$ seconds for all runs of DP-UCT, where n is the number of blocks in the problem. We use Simple ASNets with the influence constant M set to 10, 50 and 100 to demonstrate how DP-UCT can overcome the misleading information provided by the ASNet. We do not run experiments that use ASNets as a simulation function, as that would result in completely misleading state-value and action-value estimates in the search tree, meaning DP-UCT would achieve near-zero coverage.

Figure 3 depicts our results. ASNets achieves zero coverage, while UCT* is able to reliably achieve near-full coverage for all problems up to 20 blocks. In general, as we increase M, the coverage of Simple ASNets decays earlier as the number of blocks increases. This is not unexpected, as by increasing M, we increasingly 'push' the UCB1 term to select actions that the ASNet wishes to exploit, and hence misguide the navigation of the search space. Nevertheless, Simple ASNets is able to achieve near-full coverage for problems with up to 17 blocks for $M = 10$, 15 blocks for $M = 50$, and approximately 11 blocks for $M = 100$. We also observed a general increase in the time taken to reach a goal as we increased M, though this was not always the case due to the noise of DP-UCT.

This experiment shows that Simple ASNets is capable of *learning what ASNet has not learned* and being *robust to changes in the environment* by correcting the bad actions the ASNet suggests through search and eventually converging to the optimal solution.

Exploding Blocksworld. For all DP-UCT flavors, we increased the UCB1 bias parameter B to 4 and set the maximum number of trials to 30,000 in order to promote more exploration. To combine DP-UCT with ASNets, we use Ranked ASNets with the influence constant M set to 10, 50 and 100. Note, that the coverage for Exploding Blocksworld is an approximation of the true probability of reaching the goal. Since we only run each algorithm 30 times, the results are susceptible to chance.

Table 1 shows our results.[2] Since the training set used by ASNets was likely not representative of the evaluation problems (i.e., the IPPC 2008 problems), the policy learned by ASNets is suboptimal and failed to to reach the goal for the relatively easy problems (e.g., p04 and p07) while UCT* was able to more reliably solve these problems.

By combining DP-UCT with ASNets through Ranked ASNets, we were able to either match the performance of UCT* or outperform it, even when ASNet achieved zero coverage for the given problem. However, for certain configurations, we were able to improve upon all other configurations. For p08, Ranked ASNets with $M = 50$ achieves a coverage of 10/30, while all other configurations of DP-UCT are only able to achieve a coverage of around 4/30. Despite the fact that the ASNet achieved zero coverage in this experiment, the general knowledge learned by the ASNet helped us navigate the search tree more effectively and efficiently, even if the suggestions provided by the ASNet are not optimal. The same reasoning applies to the results for p04, where Ranked ASNets with $M = 50$ achieves a higher coverage than all other configurations.

We have demonstrated that we can exploit the policy learned by an ASNet to achieve more promising results than UCT* and the network itself, even if this policy is suboptimal. Thus, we have shown that Ranked ASNets is capable of *learning what the ASNet has not learned* and *improving the suboptimal policy* learned by the network.

CosaNostra Pizza. For this experiment, we considered ASNets as both a simulation function (Stochastic and Maximum ASNets), and in the UCB1 term for action selection (Simple and Ranked ASNets with $M = 100$) to improve upon UCT*. The optimal policy for CosaNostra Pizza takes $3n + 4$ steps, where n is the number of toll booths in the problem. We set the trial length when using ASNets as a simulation function to be $\lfloor 1.25 \cdot (3n + 4) \rfloor$, where the 25% increase gives some leeway for Stochastic ASNets.

Figure 4 shows our results – the curves for ASNets and Maximum ASNets overlap, as well as the curves for Simple and Ranked ASNets. ASNets achieves full coverage for all problems, while UCT* alone is only able to achieve full coverage for the problems with 2 and 3 toll booths. Using ASNets in the action selection ingredient through Simple or Ranked ASNets with the influence constant $M = 100$ only allows us to additionally achieve full coverage for the problem with 4 toll booths. This is because Simple and Ranked ASNets guide the action selection towards the optimal action, but UCT still forces the exploration of other parts of the state space.

We are able to more reliably solve CosaNostra Pizza prob-

[2] Since the difficulty of Exploding Blocksworld instances does not increase monotonically with problem size, presenting the results as a plot can be misleading.

Planner/Prob.	p01	p02	p03	p04	p05	p06	p07	p08
ASNets	16/30 8.0 ± 0.0 0.18 ± 0.14s	10/30 12.0 ± 0.0 0.17 ± 0.01s	6/30 10.0 ± 0.0 0.2 ± 0.04s	-	30/30 6.0 ± 0.0 0.19 ± 0.07s	19/30 12.0 ± 0.0 0.42 ± 0.12s	-	-
UCT*	26/30 10.92 ± 0.52 102.51 ± 5.24s	9/30 18.22 ± 1.62 175.01 ± 16.24s	13/30 25.23 ± 8.86 222.27 ± 88.77s	11/30 14.55 ± 0.63 136.46 ± 6.75s	30/30 6.13 ± 0.19 36.51 ± 2.4s	28/30 13.93 ± 0.8 132.36 ± 8.11s	30/30 13.0 ± 0.73 107.11 ± 6.95s	5/30 36.4 ± 5.09 335.87 ± 54.56s
Ranked ASNets $M = 10$	25/30 10.96 ± 0.48 100.21 ± 6.01s	6/30 17.0 ± 3.45 164.77 ± 34.89s	11/30 30.0 ± 13.64 280.25 ± 135.07s	10/30 14.4 ± 0.6 125.74 ± 11.93s	30/30 6.0 ± 0.0 38.11 ± 1.17s	25/30 13.6 ± 0.83 113.56 ± 8.11s	30/30 12.07 ± 0.14 116.36 ± 1.4s	4/30 35.0 ± 7.58 340.82 ± 75.18s
Ranked ASNets $M = 50$	23/30 11.04 ± 0.58 94.17 ± 6.51s	10/30 17.6 ± 2.85 166.29 ± 27.91s	14/30 35.71 ± 7.87 352.14 ± 78.66s	15/30 14.4 ± 0.46 123.06 ± 5.75s	30/30 6.0 ± 0.0 38.85 ± 1.15s	27/30 13.33 ± 0.76 127.69 ± 7.59s	30/30 12.07 ± 0.14 102.57 ± 1.38s	10/30 38.6 ± 0.97 374.93 ± 12.01s
Ranked ASNets $M = 100$	25/30 11.04 ± 0.48 105.26 ± 4.83s	12/30 17.33 ± 2.44 167.75 ± 24.5s	14/30 28.43 ± 6.54 259.18 ± 65.16s	10/30 14.6 ± 0.69 126.61 ± 6.41s	30/30 6.0 ± 0.0 39.41 ± 1.08s	29/30 13.38 ± 0.74 111.66 ± 7.15s	30/30 12.33 ± 0.28 103.56 ± 3.16s	4/30 36.5 ± 9.14 344.06 ± 93.88s

Table 1: Results for Exploding Blocksworld. The coverage (i.e., the number of runs that successfully reached the goal) is presented in the 1st line of each cell. The 2nd and 3rd lines of each cell show the mean cost and mean time to reach a goal, respectively, and their associated 95% confidence interval.

lems when using ASNets as a simulation function. Since the ASNet learns the optimal policy, an ASNet-based simulation function allow us to obtain much better state-value estimates for nodes in the search tree than those provided by a domain-independent heuristic. It is easy to see that when we use Maximum ASNets, the state-value $V^*(n_d)$ for the tip node of a trial n_d obtained from the simulation is optimal (assuming a sufficiently large trial length). Thus, Maximum ASNets achieves full coverage for all problems as Maximum ASNets will always provide DP-UCT with a path directly to the goal which it will eventually fall back to. For Stochastic ASNets, we see an exponential decay in the coverage as the problem size increases above 10 toll booths. The reason for this is because as the problem size increases, the probability of obtaining a path that leads directly to the goal decreases as the state space increases exponentially. Hence, DP-UCT cannot fall back to the path the ASNet has provided it, as this path may not have been taken before.

The explanations above also justify why Maximum ASNets took less time to reach a goal than all other configurations of DP-UCT. For this same reason, Maximum ASNets took less time to reach a goal than all other configurations of DP-UCT, e.g., for $n = 4$, the mean time to reach a goal and the 95% confidence interval for the considered planners are: ASNets (0.15 ± 0.05s), UCT* (64.95 ± 7.16s), Maximum ASNets (54.51 ± 0.19s), Stochastic ASNets (64.7 ± 3.17s), Simple ASNets with $M = 100$ (104.45 ± 2.38s), Ranked ASNets with $M = 100$ (124.41 ± 7.27s).

In this experiment, we have shown how using ASNets in UCB1 through SIMPLE-ASNET or RANKED-ASNET action selection can only provide marginal improvements over UCT* when the number of reachable states increases exponentially with the problem size, and the heuristic estimates are misleading. We also demonstrated how we can combat this *sub-optimal* performance of DP-UCT by using ASNets as a simulation function, as it allows us to more efficiently explore the search space and find the optimal actions. Thus, an ASNet-based simulation function may help DP-UCT *learn what it has not learned*.

Triangle Tireworld (Little and Thiébaux 2007). Triangle Tireworld is a domain with avoidable dead ends. ASNets is trivially able to find the optimal policy which always avoids dead ends. The results of our new algorithms on Triangle Tireworld are very similar to the results in the CosaNostra experiments, as the algorithms leverage the fact that ASNets finds the optimal generalized policy for both domains.

6 Conclusion, Related and Future Work

In this paper, we have investigated techniques to improve search using generalized policies. We discussed a framework for DP-UCT, extended from THTS, that allowed us to generate different flavors of DP-UCT including those that exploited the generalized policy learned by an ASNet. We then introduced methods of using this generalized policy in the simulation function, through STOCHASTIC ASNETS and MAXIMUM ASNETS. These allowed us to obtain more accurate state-value estimates and action-value estimates in the search tree. We also extended UCB1 to bias the navigation of the search space to the actions that an ASNet wants to exploit whilst maintaining the fundamental balance between exploration and exploitation, by introducing SIMPLE-ASNET and RANKED-ASNET action selection.

We have demonstrated through our experiments that our algorithms are capable of improving the capability of an ASNet to generalize beyond the distribution of problems it was trained on, as well as improve sub-optimal learning. By combining DP-UCT with ASNets, we are able to bias the exploration of actions to those that an ASNet wishes to exploit, and allow DP-UCT to converge to the optimal action in a smaller number of trials. Our experiments have also demonstrated that by harnessing the power of search, we may overcome any misleading information provided by an ASNet due to a change in the environment. Hence, we achieved the three following goals: (1) *Learn what we have not learned*, (2) *Improve on sub-optimal learning*, and (3) *Be robust to changes in the environment or domain*.

It is important to observe that our contributions are more generally applicable to any method of learning a (generalized) policy (not just ASNets), and potentially to other trial-

based search algorithms including (L)RTDP.

In the deterministic setting, there has been a long tradition of learning generalized policies and using them to guide heuristic Best First Search (BFS). For instance, Yoon et al. (Yoon, Fern, and Givan 2007) add the states resulting from selecting actions prescribed by the learned generalized policy to the the queue of a BFS guided by a relaxed-plan heuristic, and de la Rosa et al. (2011) learn and use generalized policies to generate lookahead states within a BFS guided by the FF heuristic. These authors observe that generalized policies provide effective search guidance, and that search helps correcting deficiencies in the learned policy. Search control knowledge à la TLPlan, Talplanner or SHOP2 has been successfully used to prune the search of probabilistic planners (Kuter and Nau 2005; Thiébaux et al. 2006). More recently, Steinmetz et al. (2016) have also experimented with the use of preferred actions in variants of RTDP (Barto, Bradtke, and Singh 1995) and AO* (Nilsson 1980), albeit with limited success. Our work differs from these approaches by focusing explicitly on MCTS as the search algorithm and, unlike existing work combining deep learning and MCTS (e.g. AlphaGo (Silver et al. 2016)), looks not only at using neural network policies as a simulation function for rollouts, but also as a means to bias the UCB1 action selection rule.

There are still many potential avenues for future work. We may investigate how to automatically learn the influence parameter M for SIMPLE-ASNET and RANKED-ASNET action selection, or how to combat bad information provided by an ASNet in a simulation function by mixing ASNet simulations with random simulations. We may also investigate techniques to interleave planning with learning by using UCT with ASNets as a 'teacher' for training an ASNet, similar to the 'leapfrogging' idea presented by Groshev et al. (2018). ASNets may also be replaced by Deep Reactive Policies (Issakkimuthu, Fern, and Tadepalli 2018; Bajpai, Garg, and Mausam 2018), which learn reactive policies for RDDL problems. We hope that such work would bridge the gap between symbolic AI and deep learning, and improve the state-of-the-art in probabilistic planning.

References

Bajpai, A.; Garg, S.; and Mausam. 2018. Transfer of Deep Reactive Policies for MDP Planning. In *NeurIPS*.

Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to Act using Real-Time Dynamic Programming. *Artificial intelligence* 72(1-2):81–138.

Bertsekas, D. P., and Tsitsiklis, J. N. 1991. An Analysis of Stochastic Shortest Path Problems. *Mathematics of Operations Research* 16(3):580–595.

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

Bonet, B., and Geffner, H. 2003. Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In *ICAPS*.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43.

Bryce, D., and Buffet, O. 2008. 6th International Planning Competition: Uncertainty Track. In *Proc. 3rd Int. Probabilistic Planning Competition*.

de la Rosa, T.; Celorrio, S. J.; Fuentetaja, R.; and Borrajo, D. 2011. Scaling up Heuristic Planning with Relational Decision Trees. *J. Artif. Intell. Res.* 40:767–813.

Groshev, E.; Tamar, A.; Goldstein, M.; Srivastava, S.; and Abbeel, P. 2018. Learning Generalized Reactive Policies using Deep Neural Networks. In *AAAI Spring Symposium*.

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

Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems. In *ICAPS*.

Keller, T., and Helmert, M. 2013. Trial-Based Heuristic Tree Search for Finite Horizon MDPs. In *ICAPS*.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo Planning. In *ECML*. Springer.

Kolobov, A.; Mausam; and Weld, D. S. 2012. A Theory of Goal-oriented MDPs with Dead Ends. In *UAI*.

Kuter, U., and Nau, D. S. 2005. Using Domain-Configurable Search Control for Probabilistic Planning. In *AAAI*.

Little, I., and Thiébaux, S. 2007. Probabilistic Planning vs. Replanning. In *ICAPS Workshop on IPC: Past, Present and Future*.

Nilsson, N. J. 1980. *Principles of Artificial Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of Go with deep neural networks and tree search. *nature* 529(7587):484.

Steinmetz, M.; Hoffmann, J.; and Buffet, O. 2016. Goal Probability Analysis in Probabilistic Planning: Exploring and Enhancing the State of the Art. *J. Artif. Intell. Res.* 57:229–271.

Teichteil-Königsbuch, F.; Vidal, V.; and Infantes, G. 2011. Extending Classical Planning Heuristics to Probabilistic Planning with Dead-Ends. In *AAAI*.

Thiébaux, S.; Gretton, C.; Slaney, J. K.; Price, D.; and Kabanza, F. 2006. Decision-Theoretic Planning with non-Markovian Rewards. *J. Artif. Intell. Res.* 25:17–74.

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action Schema Networks: Generalised Policies with Deep Learning. In *AAAI*.

Trevizan, F. W.; Thiébaux, S.; and Haslum, P. 2017. Occupation Measure Heuristics for Probabilistic Planning. In *ICAPS*.

Yoon, S. W.; Fern, A.; and Givan, R. 2007. Using Learned Policies in Heuristic-Search Planning. In *IJCAI*.

Pattern Selection for Optimal Classical Planning with Saturated Cost Partitioning

Jendrik Seipp
University of Basel
Basel, Switzerland
jendrik.seipp@unibas.ch

Abstract

Pattern databases are the foundation of some of the strongest admissible heuristics for optimal classical planning. Experiments showed that the most informative way of combining information from multiple pattern databases is to use saturated cost partitioning. Previous work selected patterns and computed saturated cost partitionings over the resulting pattern database heuristics in two separate steps. We introduce a new method that uses saturated cost partitioning to select patterns and show that it outperforms all existing pattern selection algorithms.

1 Introduction

A* search (Hart, Nilsson, and Raphael 1968) with an admissible heuristic (Pearl 1984) is one of the most successful methods for solving classical planning tasks optimally. An important building block of some of the strongest admissible heuristics are pattern database (PDB) heuristics. A PDB heuristic precomputes all goal distances in a simplified state space obtained by projecting the task to a subset of state variables, the *pattern*, and uses these distances as lower bounds on the true goal distances. PDB heuristics were originally introduced for solving the 15-puzzle (Culberson and Schaeffer 1996) and have later been generalized to many other combinatorial search tasks (e.g., Korf 1997; Felner, Korf, and Hanan 2004) and to the setting of domain-independent planning (Edelkamp 2001).

Using a single PDB heuristic of reasonable size is usually not enough to cover sufficiently many aspects of challenging planning tasks. It is therefore often beneficial to compute multiple PDB heuristics and to combine their estimates admissibly (Holte et al. 2006). The simplest approach for this is to choose the PDB with the highest estimate in each state. Instead of this maximization scheme, we would like to sum estimates, but this renders the resulting heuristic inadmissible in general. However, if two PDBs are affected by disjoint sets of operators, they are *independent* and we can admissibly add their estimates (Korf and Felner 2002; Felner, Korf, and Hanan 2004). Haslum et al. (2007) later generalized this idea by introducing the *canonical heuristic* for PDBs, which computes all maximal subsets of pairwise independent PDBs and then uses the maximum over the sums of independent PDBs as the heuristic value.

Cost partitioning (Katz and Domshlak 2008; Yang et al. 2008) is a generalization of the independence-based methods above. It makes the sum of heuristic estimates admissible by distributing the costs of each operator among the heuristics. The literature contains many different cost partitioning algorithms such as zero-one cost partitioning (Edelkamp 2002; Haslum et al. 2007), uniform cost partitioning (Katz and Domshlak 2008), optimal cost partitioning (Katz and Domshlak 2008; Karpas and Domshlak 2009; Katz and Domshlak 2010; Pommerening et al. 2015), post-hoc optimization (Pommerening, Röger, and Helmert 2013) and delta cost partitioning (Fan, Müller, and Holte 2017).

In previous work (Seipp, Keller, and Helmert 2017a), we showed experimentally for the benchmark tasks from previous International Planning Competitions (IPC) that *saturated cost partitioning* (SCP) (Seipp and Helmert 2014; 2018) is the cost partitioning algorithm of choice for PDB heuristics. Saturated cost partitioning considers an ordered sequence of heuristics. Iteratively, it gives each heuristic the minimum amount of costs that the heuristic needs to justify all its estimates and then uses the remaining costs for subsequent heuristics until all heuristics have been served this way.

Before we can compute a saturated cost partitioning over pattern database heuristics, we need to select a collection of patterns. The first domain-independent automated pattern selection algorithm is due to Edelkamp (2001). It partitions the state variables into patterns via best-fit bin packing. Edelkamp (2006) later used a genetic algorithm to search for a pattern collection that maximizes the average heuristic value of a zero-one cost partitioning over the PDB heuristics.

Haslum et al. (2007) proposed an algorithm that performs a hill-climbing search in the space of pattern collections (HC). HC evaluates a collection C by estimating the search effort of the canonical heuristic over C based on a model of IDA* runtime (Korf, Reid, and Edelkamp 2001).

Franco et al. (2017) presented the Complementary PDBs Creation (CPC) method, that combines bin packing and genetic algorithms to create a pattern collection minimizing the estimated search effort of an A* search (Lelis, Stern, and Sturtevant 2014).

Rovner, Sievers, and Helmert (2019) repeatedly compute patterns using counterexample-guided abstraction refinement (CEGAR): starting from a random goal variable,

their CEGAR algorithm iteratively finds solutions in the corresponding projection and executes them in the original state space. Whenever a solution cannot be executed due to a violated precondition, it adds the missing precondition variable to the pattern.

Finally, Pommerening, Röger, and Helmert (2013) systematically generate all *interesting* patterns up to a given size X (SYS-X). Experiments showed that cost-partitioned heuristics over SYS-2 and SYS-3 yield accurate estimates (Pommerening, Röger, and Helmert 2013; Seipp, Keller, and Helmert 2017a), but using all interesting patterns of larger sizes is usually infeasible.

We introduce SYS-SCP, a new pattern selection algorithm based on saturated cost partitioning that potentially considers all interesting patterns, but only selects useful ones. SYS-SCP builds multiple pattern sequences that together form the resulting pattern collection. For each sequence σ, it considers the interesting patterns in increasing order by size and adds a pattern P to σ if P is not part of an earlier sequence and the saturated cost partitioning heuristic over σ plus P is more informative than the one over σ alone.

2 Background

We consider optimal classical planning tasks in a SAS$^+$-like notation (Bäckström and Nebel 1995) and represent a planning task Π as a tuple $\langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$. Each variable v in the finite set of variables $\mathcal{V}$ has a finite domain $\mathit{dom}(v)$. A *partial state* s is defined over a subset of variables $\mathit{vars}(s) \subseteq \mathcal{V}$ and maps each $v \in \mathit{vars}(s)$ to a value in $\mathit{dom}(v)$, written as $s[v]$. We call the pair $\langle v, s[v] \rangle$ an *atom* and interchangeably treat partial states as mappings from variables to values or as sets of atoms. If $\mathit{vars}(s) = \mathcal{V}$, we call s a *state*. We write $S(\Pi)$ for the set of all states in Π.

Each operator o in the finite set of operators $\mathcal{O}$ has a *precondition* $\mathit{pre}(o)$ and an *effect* $\mathit{eff}(o)$, both of which are partial states, and a cost $\mathit{cost}(o) \in \mathbb{R}_0^+$. An operator o is applicable in a state s if $\mathit{pre}(o) \subseteq s$. Applying o in s leads into state $s' = s[\![o]\!]$ with $s'[v] = \mathit{eff}(o)[v]$ for all $v \in \mathit{vars}(\mathit{eff}(o))$ and $s'[v] = s[v]$ for all variables $v \in \mathcal{V} \setminus \mathit{vars}(\mathit{eff}(o))$. The state s_0 is called the *initial state* and $s_\star$ is a partial state, the *goal*.

Transition systems assign semantics to planning tasks.

Definition 1 (Transition Systems)**.** *A* transition system $\mathcal{T}$ *is a labeled digraph defined by a finite set of* states $S(\mathcal{T})$*, a finite set of* labels $L(\mathcal{T})$*, a set* $T(\mathcal{T})$ *of labeled* transitions $s \xrightarrow{\ell} s'$ *with* $s, s' \in S(\mathcal{T})$ *and* $\ell \in L(\mathcal{T})$*, an* initial state $s_0(\mathcal{T})$*, and a set* $S_\star(\mathcal{T})$ *of* goal states*.*

A planning task $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$ induces a transition system $\mathcal{T}$ with states $S(\Pi)$, labels $\mathcal{O}$, transitions $\{s \xrightarrow{o} s[\![o]\!] \mid s \in S(\Pi), o \in \mathcal{O}, \mathit{pre}(o) \subseteq s\}$, initial state s_0 and goal states $\{s \in S(\Pi) \mid s_\star \subseteq s\}$.

Separating transition systems from *cost functions* allows us to evaluate the same transition system under different cost functions, which is important for cost partitioning.

Definition 2 (Cost Functions)**.** *A* cost function *for transition system* $\mathcal{T}$ *is a function* $\mathit{cost} : L(\mathcal{T}) \to \mathbb{R} \cup \{-\infty, \infty\}$*. It is* finite *if* $-\infty < \mathit{cost}(\ell) < \infty$ *for all labels* ℓ*. It is* non-negative *if* $\mathit{cost}(\ell) \geq 0$ *for all labels* ℓ*. We write* $\mathcal{C}(\mathcal{T})$ *for the set of all cost functions for* $\mathcal{T}$*.*

Note that we assume that the cost function of the planning task is non-negative and finite, but as in previous work we allow negative (Pommerening et al. 2015) and infinite costs (Seipp and Helmert 2019) in cost partitionings. The generalization to infinite costs is necessary to cleanly state some of our definitions.

Definition 3 (Weighted Transition Systems)**.** *A* weighted transition system *is a pair* $\langle \mathcal{T}, \mathit{cost} \rangle$ *where* $\mathcal{T}$ *is a transition system and* $\mathit{cost} \in \mathcal{C}(\mathcal{T})$ *is a cost function for* $\mathcal{T}$*.*

The *cost* of a *path* $\pi = \langle s^0 \xrightarrow{\ell_1} s^1, \ldots, s^{n-1} \xrightarrow{\ell_n} s^n \rangle$ in a weighted transition system $\langle \mathcal{T}, \mathit{cost} \rangle$ is defined as $\mathit{cost}(\pi) = \sum_{i=1}^{n} \mathit{cost}(\ell_i)$. It is ∞ if the sum contains both $+\infty$ and $-\infty$. If s^n is a goal state, π is called a *goal path* for s^0.

Definition 4 (Goal Distances and Optimal Paths)**.** *The* goal distance *of a state* $s \in S(\mathcal{T})$ *in a weighted transition system* $\langle \mathcal{T}, \mathit{cost} \rangle$ *is defined as* $\inf_{\pi \in \Pi_\star(\mathcal{T}, s)} \mathit{cost}(\pi)$*, where* $\Pi_\star(\mathcal{T}, s)$ *is the set of goal paths from* s *in* $\mathcal{T}$*. (The infimum of the empty set is* ∞*.) We write* $h^*_{\mathcal{T}}(\mathit{cost}, s)$ *for the goal distance of* s*. If* $h^*_{\mathcal{T}}(\mathit{cost}, s) = \infty$*, we call* s unsolvable*. A goal path* π *from* s *is* optimal *if* $\mathit{cost}(\pi) = h^*_{\mathcal{T}}(\mathit{cost}, s)$*.*

Optimal classical planning is the problem of finding an optimal goal path from s_0 or showing that s_0 is unsolvable.

We use *heuristics* to estimate goal distances (Pearl 1984).

Definition 5 (Heuristics)**.** *A* heuristic *for a transition system* $\mathcal{T}$ *is a function* $h : \mathcal{C}(\mathcal{T}) \times S(\mathcal{T}) \to \mathbb{R} \cup \{-\infty, \infty\}$*. Heuristic* h *is* admissible *if* $h(\mathit{cost}, s) \leq h^*_{\mathcal{T}}(\mathit{cost}, s)$ *for all* $\mathit{cost} \in \mathcal{C}(\mathcal{T})$ *and all* $s \in S(\mathcal{T})$*.*

Cost partitioning makes adding heuristics admissible by distributing the costs of each operator among the heuristics.

Definition 6 (Cost Partitioning)**.** *Let* $\mathcal{T}$ *be a transition system. A* cost partitioning *for a cost function* $\mathit{cost} \in \mathcal{C}(\mathcal{T})$ *is a tuple* $\langle \mathit{cost}_1, \ldots, \mathit{cost}_n \rangle \in \mathcal{C}(\mathcal{T})^n$ *whose sum is bounded by cost:* $\sum_{i=1}^{n} \mathit{cost}_i(\ell) \leq \mathit{cost}(\ell)$ *for all* $\ell \in L(\mathcal{T})$*. A cost partitioning* $\langle \mathit{cost}_1, \ldots, \mathit{cost}_n \rangle \in \mathcal{C}(\mathcal{T})^n$ *over the heuristics* $\langle h_1, \ldots, h_n \rangle$ *for* $\mathcal{T}$ *induces the* cost-partitioned heuristic $h(\mathit{cost}, s) = \sum_{i=1}^{n} h_i(\mathit{cost}_i, s)$*. If the sum contains* $+\infty$ *and* $-\infty$*, it evaluates to the leftmost infinite value.*

One of the cost partitioning algorithms from the literature is *saturated cost partitioning* (Seipp and Helmert 2018). It is based on the insight that we can often reduce the amount of costs given to a heuristic without changing any heuristic estimates. *Saturated cost functions* formalize this idea.

Definition 7 (Saturated Cost Function)**.** *Consider a transition system* $\mathcal{T}$*, a heuristic* h *for* $\mathcal{T}$ *and a cost function* $\mathit{cost} \in \mathcal{C}(\mathcal{T})$*. A cost function* $\mathit{scf} \in \mathcal{C}(\mathcal{T})$ *is* saturated *for* h *and cost if*

1. $\mathit{scf}(\ell) \leq \mathit{cost}(\ell)$ *for all labels* $\ell \in L(\mathcal{T})$ *and*
2. $h(\mathit{scf}, s) = h(\mathit{cost}, s)$ *for all states* $s \in S(\mathcal{T})$*.*

A saturated cost function scf *is* minimal *if there is no other saturated cost function* scf' *for* h *and cost with* $\mathit{scf}(\ell) \leq \mathit{scf}'(\ell)$ *for all labels* $\ell \in L(\mathcal{T})$*.*

Whether we can efficiently compute a minimal saturated cost function depends on the type of heuristic. In earlier work (Seipp and Helmert 2018), we showed that this is possible for explicitly-represented abstraction heuristics (Helmert, Haslum, and Hoffmann 2007), which include PDB heuristics.

Definition 8 (Minimum Saturated Cost Function for Abstraction Heuristics)**.** *Let $\langle \mathcal{T}, cost \rangle$ be a weighted transition system and h an abstraction heuristic for $\mathcal{T}$ with abstract transition system $\mathcal{T}'$. The* minimum saturated cost function *mscf for h and cost is*

$$mscf(\ell) = \sup_{a \xrightarrow{\ell} b \in T(\mathcal{T}')} (h^*_{\mathcal{T}'}(cost, a) - h^*_{\mathcal{T}'}(cost, b))$$

for all $\ell \in L(\mathcal{T})$, where $x - y = -\infty$ iff $x = -\infty$ or $y = \infty$.

Given a sequence of abstraction heuristics, the saturated cost partitioning algorithm iteratively assigns to each heuristic only the costs that the heuristic needs to preserve its estimates and uses the remaining costs for subsequent heuristics.

Definition 9 (Saturated Cost Partitioning)**.** *Consider a transition system $\mathcal{T}$ and a sequence of abstraction heuristics $\mathcal{H} = \langle h_1, \ldots, h_n \rangle$ for $\mathcal{T}$. For all $1 \leq i \leq n$, saturate$_i$: $\mathcal{C}(\mathcal{T}) \to \mathcal{C}(\mathcal{T})$ receives a cost function rem and returns the minimum saturated cost function for h_i and rem. The* saturated cost partitioning $\langle cost_1, \ldots, cost_n \rangle$ *of a function $cost \in \mathcal{C}(\mathcal{T})$ over $\mathcal{H}$ is defined as:*

$$\begin{aligned} rem_0 &= cost \\ cost_i &= saturate_i(rem_{i-1}) && \text{for all } 1 \leq i \leq n \\ rem_i &= rem_{i-1} - cost_i && \text{for all } 1 \leq i \leq n, \end{aligned}$$

where the auxiliary cost functions rem$_i$ represent the remaining costs after processing the first i heuristics in $\mathcal{H}$.

We write $h^{\text{SCP}}_{\mathcal{H}}$ for the saturated cost partitioning heuristic over the sequence of heuristics $\mathcal{H}$. In this work, we compute saturated cost partitionings over pattern database heuristics.

A *pattern* for task Π with variables $\mathcal{V}$ is a subset $P \subseteq \mathcal{V}$. By syntactically removing all variables from Π that are not in P, we obtain the *projected* task $\Pi|_P$ inducing the abstract transition system $\mathcal{T}_P$. The PDB heuristic h^P for a pattern P is defined as $h^P(cost, s) = h^*_{\mathcal{T}_P}(cost, s|_P)$, where $s|_P$ is the abstract state that s is projected to in $\Pi|_P$. For the pattern sequence $\langle P_1, \ldots, P_n \rangle$ we define $h^{\text{SCP}}_{\langle P_1, \ldots, P_n \rangle} = h^{\text{SCP}}_{\langle h^{P_1}, \ldots, h^{P_n} \rangle}$.

One of the simplest pattern selection algorithms is to generate all patterns up to a given size X (Felner, Korf, and Hanan 2004) and we call this approach SYS-NAIVE-X. It is easy to see that for tasks with n variables, SYS-NAIVE-X generates $\sum_{i=1}^{X} \binom{n}{i}$ patterns. Usually, many of these patterns do not add much information to a cost-partitioned heuristic over the patterns. Unfortunately, there is no efficiently computable test that allows us to discard such uninformative patterns. Even patterns without any goal variables can increase heuristic estimates in a cost partitioning (Pommerening 2017).

However, in the setting where only non-negative cost functions are allowed in cost partitionings, there are efficiently computable criteria for deciding whether a pattern is *interesting*, i.e., whether it cannot be replaced by a set of smaller patterns that together yield the same heuristic estimates (Pommerening, Röger, and Helmert 2013).

Algorithm 1 SYS-SCP: Given a planning task with states $S(\mathcal{T})$, cost function *cost* and interesting patterns SYS, select a subset $C \subseteq$ SYS.

```
1:  function SYS-SCP(Π)
2:      C ← ∅
3:      repeat for at most T_x seconds
4:          σ ← ⟨⟩
5:          for P ∈ ORDER(SYS) and at most T_y seconds do
6:              if P ∉ C and PATTERNUSEFUL(σ, P) then
7:                  σ ← σ ⊕ P
8:                  C ← C ∪ {P}
9:      until σ = ⟨⟩
10:     return C

11: function PATTERNUSEFUL(σ, P)
12:     return ∃s ∈ S(T) :
            h^SCP_σ(cost, s) < h^SCP_{σ⊕P}(cost, s) < ∞
```

The criteria are based on the *causal graph CG*(Π) of a task Π (Helmert 2004). *CG*(Π) is a directed graph with a node for each variable in Π. If there is an operator with a precondition on u and an effect on $v \neq u$, *CG*(Π) contains a *precondition* arc from u to v. If an operator affects both u and v, *CG*(Π) contains *co-effect* arcs from u to v and from v to u.

Definition 10 (Interesting Patterns)**.** *A pattern P is* interesting *if*

1. *CG*($\Pi|_P$) *is weakly connected, and*
2. *CG*($\Pi|_P$) *contains a directed path via precondition arcs from each node to some goal variable node.*

The systematic pattern generation method SYS-X generates all interesting patterns up to size X. We let SYS denote the set of all interesting patterns for a given task. On IPC benchmark tasks, SYS-X often generates much fewer patterns than SYS-NAIVE-X for the same size limit X. Still, it is usually infeasible to compute all SYS-X patterns and the corresponding projections for $X > 3$ within reasonable amounts of time and memory. Also, we hypothesize that even when considering only interesting patterns, usually only a small percentage of the systematic patterns up to size 3 contribute much information to the resulting heuristic.

For these two reasons we propose a new pattern selection algorithm that potentially considers all interesting patterns, but only selects the ones that it deems useful.

3 Sys-SCP Pattern Selection Algorithm

Our new pattern selection algorithm repeatedly creates a new empty pattern sequence σ and only appends those interesting patterns to σ that increase any finite heuristic values of a saturated cost partitioning heuristic computed over σ.

Algorithm 1 shows pseudo-code for the procedure, which we call SYS-SCP. It starts with an empty pattern collection C. In each iteration of the outer loop, SYS-SCP creates a

new empty pattern sequence σ and then loops over the interesting patterns $P \in$ SYS in the order chosen by ORDER (see Section 3.2) for at most T_y seconds. SYS-SCP appends a pattern P to σ and includes it in C if there is a state s for which the saturated cost partitioning over σ extended by P has a higher finite heuristic value than the one over σ alone. Once an iteration selects no new patterns or SYS-SCP hits the time limit T_x, the algorithm stops and returns C.

We impose a time limit T_x on the outer loop of the algorithm since the number of interesting patterns is exponential in the number of variables and therefore SYS-SCP usually cannot evaluate them all in a reasonable amount of time. By imposing a time limit T_y on the inner loop, we allow SYS-SCP to periodically start over with a new empty pattern sequence.

The most important component of the SYS-SCP algorithm is the PATTERNUSEFUL function that decides whether to select a pattern P. The function enumerates all states $s \in S(\Pi)$, which is obviously infeasible for all but the smallest tasks Π. Fortunately, we can efficiently compute an equivalent test in the projection to P.

Lemma 1. *Consider a planning task Π with non-negative cost function cost and induced transition system $\mathcal{T}$. Let $s \in S(\mathcal{T})$ be a state, P be a pattern for Π and σ be a (possibly empty) sequence of patterns $\langle P_1, \ldots, P_n \rangle$ for Π. Finally, let rem be the remaining cost function after computing h^{SCP}_σ for cost.*

$$h^{SCP}_\sigma(cost, s) < h^{SCP}_{\sigma \oplus P}(cost, s) < \infty$$
$$\Leftrightarrow\ 0 < h^*_{\mathcal{T}_P}(rem, s|_P) < \infty$$

Proof. $h^{\text{SCP}}_\sigma(cost, s) < h^{\text{SCP}}_{\sigma \oplus P}(cost, s) < \infty$

$$\overset{(1)}{\Leftrightarrow} h^{\text{SCP}}_{\langle P_1, \ldots, P_n \rangle}(cost, s) < h^{\text{SCP}}_{\langle P_1, \ldots, P_n, P \rangle}(cost, s) < \infty$$
$$\overset{(2)}{\Leftrightarrow} \sum_{i=1}^{n} h^{P_i}(cost_i, s) < \sum_{i=1}^{n} h^{P_i}(cost_i, s) + h^P(rem, s) < \infty$$
$$\overset{(3)}{\Leftrightarrow} 0 < h^P(rem, s) < \infty \overset{(4)}{\Leftrightarrow} 0 < h^*_{\mathcal{T}_P}(rem, s|_P) < \infty$$

Step 1 substitutes $\langle P_1, \ldots, P_n \rangle$ for σ and Step 2 uses the definition of saturated cost partitioning heuristics. For Step 3 we need to show that $x = \sum_{i=1}^{n} h^{P_i}(cost_i, s)$ is finite.

The inequality states $x < \infty$. We now show $x \geq 0$, which implies $x > -\infty$. Using requirement 1 for saturated cost functions from Definition 7 and the fact that $rem_0 = cost$ is non-negative, it is easy to see that all remaining cost functions are non-negative. Consequently, $h^{P_i}(cost_i, s) = h^{P_i}(rem_{i-1}, s) \geq 0$ for all $s \in S(\mathcal{T})$, which uses requirement 2 from Definition 7 and the fact that goal distances are non-negative in transition systems with non-negative weights.

Step 4 uses the definition of PDB heuristics. □

Theorem 1 (Computing PATTERNUSEFUL on Projections)**.** *Consider a planning task Π with non-negative cost function cost and induced transition system $\mathcal{T}$. Let P be a single pattern and σ be a (possibly empty) sequence of patterns. Finally, let rem be the remaining cost function after computing h^{SCP}_σ for cost.*

$$\exists s \in S(\mathcal{T}) : h^{SCP}_\sigma(cost, s) < h^{SCP}_{\sigma \oplus P}(cost, s) < \infty$$
$$\Leftrightarrow \exists s' \in S(\mathcal{T}_P) : 0 < h^*_{\mathcal{T}_P}(rem, s') < \infty$$

Proof. Follows directly from Lemma 1 and the fact that projections are induced abstractions: for each abstract state s' in an induced abstraction there is at least one concrete state s which is projected to s'. □

We use Theorem 1 in our SYS-SCP implementation by keeping track of the cost function *rem*, i.e., the costs that remain after computing h^{SCP}_σ. We select a pattern P if there are any goal distances d with $0 < d < \infty$ in $\mathcal{T}_P$ under *rem*.

Theorem 1 also removes the need to compute $h^{\text{SCP}}_{\sigma \oplus P}$ from scratch for every pattern P. This is important since we want to decide whether or not to add P quickly and this operation should not become slower when σ contains more patterns.

3.1 Dead Ends

To obtain high finite heuristic values for solvable states it is important to choose good cost partitionings. In contrast, cost functions are irrelevant for detecting unsolvable states. This is the underlying reason why Lemma 1 only holds for finite values and therefore why SYS-SCP ignores unsolvable states.

However, we can still use the information about unsolvable states contained in projections. It is easy to see that each abstract state in a projection corresponds to a partial state in the original task. If an abstract state is unsolvable in a projection, we call the corresponding partial state a *dead end*. Since projections preserve all paths, any state in the original task subsuming a dead end is unsolvable. We can extract all dead ends from the projections that SYS-SCP evaluates and use this information to prune unsolvable states during the A* search (Pommerening and Seipp 2016).

3.2 Ordering Patterns

We showed in earlier work that the order in which saturated cost partitioning considers the component heuristics has a strong influence on the quality of the resulting heuristic (Seipp, Keller, and Helmert 2017b). Choosing a good order is even more important for SYS-SCP, since it usually only sees a subset of interesting patterns within the allotted time. To ensure that this subset of interesting patterns covers different aspects of the planning task, we let the ORDER function generate the interesting patterns in increasing order by size.

This leaves the question how to sort patterns of the same size. We propose four methods for making this decision. The first one (*random*) simply orders patterns of the same size randomly. The remaining three assign a key to each pattern, allowing us to sort by key in increasing or decreasing order.

Causal Graph. The first ordering method is based on the insight that it is often more important to have accurate heuristic estimates near the goal states rather than elsewhere in the state space (e.g., Holte et al. 2006; Torralba,

Linares López, and Borrajo 2018). We therefore want to focus on patterns containing goal variables or variables that are closely connected to goal variables. To quantify "goal-connectedness" we use an approximate topological ordering $\prec$ of the causal graph $CG(\Pi)$. We let the function $cg : \mathcal{V} \to \mathbb{N}_0^+$ assign each variable $v \in \mathcal{V}$ to its index in $\prec$. For a given pattern P, the *cg* ordering method returns the key $\langle cg(v_1), \ldots, cg(v_n) \rangle$, where $v_i \in P$ and $cg(v_i) < cg(v_j)$ for all $1 \leq i < j \leq n$. Since the keys are unique, they define a total order. Sorting the patterns by *cg* in decreasing order (*cg-down*), yields the desired order which starts with "goal-connected" patterns.

States in Projection. Given a pattern P, the ordering method *states* returns the key $|S(\Pi|_P)|$, i.e., the number of states in the projection to P. We use *cg-down* to break ties.

Active Operators. Given a pattern P, the *ops* ordering method returns the number of operators that affect a variable in P. We break ties with *cg-down*.

4 Experiments

We implemented the SYS-SCP pattern selection algorithm in the Fast Downward planning system (Helmert 2006) and conducted experiments with the Downward Lab toolkit (Seipp et al. 2017) on Intel Xeon Silver 4114 processors. Our benchmark set consists of all 1827 tasks without conditional effects from the optimization tracks of the 1998–2018 IPCs. The tasks belong to 48 different domains. We limit time by 30 minutes and memory by 3.5 GiB. All benchmarks[1], code[2] and experimental data[3] have been published online.

To fairly compare the quality of different pattern collections, we use the same cost partitioning algorithm for all collections. Saturated cost partitioning is the obvious choice for the evaluation since experiments showed that it is preferable to all other cost partitioning algorithms for HC, SYS-2 and CPC patterns in almost all evaluated benchmark domains (Seipp, Keller, and Helmert 2017a; Rovner, Sievers, and Helmert 2019).

Diverse Saturated Cost Partitioning Heuristics. For a given pattern collection C, we compute diverse saturated cost partitioning heuristics using the diversification procedure by Seipp, Keller, and Helmert (2017b): we start with an empty family of saturated cost partitioning heuristics $\mathcal{F}$ and a set $\hat{S}$ of 1000 sample states obtained with random walks (Haslum et al. 2007). Then we iteratively sample a new state s and compute a *greedy* order ω of C that works well for s (Seipp 2017). If h^{SCP}_ω has a higher heuristic estimate for any state $s' \in \hat{S}$ than all heuristics in $\mathcal{F}$, we add h^{SCP}_ω to $\mathcal{F}$. We stop this diversification procedure after 200 seconds and then perform an A* search using the maximum over the heuristics in $\mathcal{F}$.

[1]Benchmarks: https://doi.org/10.5281/zenodo.2616479

[2]Code: https://doi.org/10.5281/zenodo.3233330

[3]Experimental data: https://doi.org/10.5281/zenodo.3233326

Coverage	10s	100s	1000s	∞
1s	**1137**	1132	1055	716
10s	1077	**1168**	1142	337
100s	1077	1082	**1154**	284
∞	1077	**1082**	989	227

Table 1: Number of tasks solved by SYS-SCP using different time limits T_x and T_y for the outer loop (x axis) and inner loop (y axis).

	cg-up	states-up	random	ops-down	states-down	ops-up	cg-down	Coverage
cg-up	–	5	6	5	4	3	3	1140.0
states-up	**6**	–	6	**8**	5	2	2	1153.0
random	**10**	**10**	–	8	7	6	3	1148.2
ops-down	**7**	**8**	**9**	–	4	7	3	1141.0
states-down	**9**	**8**	**9**	**7**	–	4	2	1152.0
ops-up	**11**	**12**	**12**	**11**	**11**	–	**6**	1166.0
cg-down	**12**	**10**	**12**	**10**	**9**	**6**	–	**1168.0**

Table 2: Per-domain coverage comparison of different orders for patterns of the same size. The entry in row r and column c shows the number of domains in which order r solves more tasks than order c. For each order pair we highlight the maximum of the entries (r, c) and (c, r) in bold. Right: Total number of solved tasks. The results for *random* are averaged over 10 runs (standard deviation: 3.36).

Before we compare SYS-SCP to other pattern selection algorithms, we evaluate the effects of changing its parameters in four ablation studies. We use at most 2M states per PDB and 20M states in the PDB collection for all SYS-SCP runs.

4.1 Time Limits

Table 1 shows that a time limit for the outer loop is more important than one for the inner loop, but for maximum coverage we need both limits. The combination that solves the highest number of tasks is 10s for the inner and 100s for the outer loop. We use these values in all other experiments.

4.2 Dead Ends

All configurations from Table 1 store the dead ends from all projections evaluated by SYS-SCP and use them to prune unsolvable states during the A* search. For the best configuration from Table 1, coverage decreases from 1168 to 1153 tasks if we ignore the dead ends. Therefore, we use dead ends for pruning unsolvable states in all other experiments.

4.3 Pattern Orders

Table 2 compares the different methods for ordering patterns of the same size. For all of *states*, *ops* and *cg*, at least one or-

Max pattern size	1	2	3	4	5
SYS-NAIVE	840	**937**	914	752	571
SYS-NAIVE-LIM	840	968	**1004**	912	878
SYS	840	986	**1057**	922	731
SYS-LIM	840	985	**1088**	1050	1035

Table 3: Number of solved tasks for naive (SYS-NAIVE) and interesting patterns (SYS). We evaluate both versions without and with time and memory limits and using different maximum pattern sizes.

dering direction (*up* or *down*) is preferable to using random orders. The *ops-up* method is preferable to *ops-down* for 11 domains, but there are also 7 domains where the opposite is the case. The relation between *states-down* and *states-up* is similar. The only ordering method where one direction is clearly preferable to the other is *cg*: *cg-down* solves more tasks than *cg-up* in 12 domains, while the opposite is the case in only 3 domains. Since *cg-down* also has the highest overall coverage, we use it in all other experiments.

4.4 Using Pattern Sequences for Diversification

Instead of discarding the computed pattern sequences when SYS-SCP finishes, we can turn each pattern sequence σ into a full pattern order by randomly appending all SYS-SCP patterns missing from σ to σ and pass the resulting order to the diversification procedure.

Feeding the diversification exclusively with such orders leads to solving 1130 tasks, while using only greedy orders for sample states (Seipp 2017) solves 1156 tasks. We obtain the best results by diversifying both types of orders, solving 1168 tasks, and we use this variant in all other experiments.

4.5 Systematic Patterns With Limits

In the next experiment, we evaluate the obvious baseline for SYS-SCP: selecting all (interesting) patterns up to a fixed size. Table 3 holds coverage results of SYS-NAIVE-X and SYS-X for $1 \leq X \leq 5$. We also include variants (*-LIM) that use at most 100 seconds, no more than 2M states in each projection and at most 20M states per collection. For the *-LIM variants, we sort the patterns in the *cg-down* order.

The results show that interesting patterns are always preferable to naive patterns, both with and without limits, which is why we only consider interesting patterns in SYS-SCP. Imposing limits is not important for SYS-1 and SYS-2, but leads to solving many more tasks for $X \geq 3$. Overall, SYS-3-LIM has the highest total coverage (1088 tasks).

4.6 Comparison of Pattern Selection Algorithms

In Table 4 we compare SYS-SCP to the strongest pattern selection algorithms from the literature: HC, SYS-3-LIM, CPC and CEGAR. (See Table 6 for per-domain coverage results.) We run each algorithm with its preferred parameter values, which implies using at most 900s for HC and CPC and 100s for the other algorithms.

HC is outperformed by all other algorithms. Interestingly, already the simple SYS-3-LIM approach is competitive with

	HC	SYS-3-LIM	CPC	CEGAR	SYS-SCP	Coverage
HC	–	8	10	8	3	966
SYS-3-LIM	**19**	–	14	10	2	1088
CPC	**20**	**15**	–	12	3	1055
CEGAR	**22**	**14**	**16**	–	3	1098
SYS-SCP	**28**	**23**	**21**	**21**	–	**1168**

Table 4: Per-domain coverage comparison of pattern selection algorithms. For an explanation of the data see the caption of Table 2.

CPC and CEGAR. However, we obtain the best results with SYS-SCP. It is preferable to all other pattern selection algorithms in per-domain comparisons: no algorithm has higher coverage than SYS-SCP in more than three domains, while SYS-SCP solves more tasks than each of the other algorithms in at least 21 domains. SYS-SCP also has the highest total coverage of 1168 tasks, solving 70 more tasks than the strongest contender. This is a considerable improvement in the setting of optimal classical planning, where task difficulty tends to scale exponentially.

4.7 Comparison to IPC Planners

In our final experiment, we evaluate whether Scorpion (Seipp 2018), one of the strongest optimal planners in IPC 2018, benefits from using SYS-SCP patterns. Scorpion computes diverse saturated cost partitioning heuristics over HC and SYS-2 PDB heuristics and Cartesian abstraction heuristics (CART) (Seipp and Helmert 2018). We abbreviate this combination with COMB=HC+SYS-2+CART. In Table 5 we compare the original Scorpion planner, three Scorpion variants that use different sets of heuristics and the top three optimal planners from IPC 2018, Delfi 1 (Sievers et al. 2019), Complementary 1 (Franco et al. 2018) and Complementary 2 (Franco et al. 2017). (Table 6 holds per-domain coverage results.) In contrast to the configurations we evaluated above, all planners in Table 5 prune irrelevant operators in a preprocessing step (Alcázar and Torralba 2015).

The results show that all Scorpion variants outperform the top three IPC 2018 planners in per-domain comparisons. We also see that Scorpion benefits from using SYS-SCP PDBs instead of the COMB heuristics in many domains. Using the union of both sets is clearly preferable to using either COMB or SYS-SCP alone, since it raises the total coverage to 1261 by 56 and 44 tasks, respectively. For maximum coverage (1265 tasks), Scorpion only needs SYS-SCP PDBs and Cartesian abstraction heuristics.

5 Conclusion

We introduced a new pattern selection algorithm based on saturated cost partitioning and showed that it outperforms

		Complementary 1	Complementary 2	Delfi 1	Scorpion: COMB	Scorpion: SYS-SCP	Scorpion: SYS-SCP+CART	Scorpion: SYS-SCP+COMB	Coverage
	Complementary 1	–	7	4	12	9	9	9	1030
	Complementary 2	**24**	–	7	12	10	9	8	1093
	Delfi 1	**35**	**28**	–	16	15	13	13	1236
Scorpion	COMB	**28**	**27**	**19**	–	7	5	2	1205
Scorpion	SYS-SCP	**29**	**25**	**21**	**15**	–	4	4	1217
Scorpion	SYS-SCP+CART	**29**	**26**	**22**	**16**	**10**	–	4	**1265**
Scorpion	SYS-SCP+COMB	**30**	**27**	**23**	**13**	**13**	**5**	–	1261

Table 5: Comparison of IPC 2018 planners and Scorpion variants.

all other pattern selection algorithms from the literature. The algorithm selects a pattern if it is useful for *any* state in the state space. In future work, we would like to evaluate whether it is beneficial to restrict this criterion to a subset of states, such as all reachable states or a set of sample states.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. We have received funding for this work from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement no. 817639).

References

Alcázar, V., and Torralba, Á. 2015. A reminder about the importance of computing and exploiting invariants in planning. In Brafman, R.; Domshlak, C.; Haslum, P.; and Zilberstein, S., eds., *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 2–6. AAAI Press.

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

Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. In McCalla, G. I., ed., *Proceedings of the Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI 1996)*, volume 1081 of *Lecture Notes in Computer Science*, 402–416. Springer-Verlag.

Edelkamp, S. 2001. Planning with pattern databases. In Cesta, A., and Borrajo, D., eds., *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, 84–90. AAAI Press.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In Ghallab, M.; Hertzberg, J.; and Traverso, P., eds., *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2002)*, 274–283. AAAI Press.

Edelkamp, S. 2006. Automated creation of pattern database search heuristics. In Edelkamp, S., and Lomuscio, A., eds., *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence (MoChArt 2006)*, 35–50.

Fan, G.; Müller, M.; and Holte, R. 2017. Additive merge-and-shrink heuristics for diverse action costs. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 4287–4293. IJCAI.

Felner, A.; Korf, R.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22:279–318.

Franco, S.; Torralba, Á.; Lelis, L. H. S.; and Barley, M. 2017. On creating complementary pattern databases. In Sierra, C., ed., *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 4302–4309. IJCAI.

Franco, S.; Lelis, L. H. S.; Barley, M.; Edelkamp, S.; Martines, M.; and Moraru, I. 2018. The Complementary1 planner in the IPC 2018. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 28–31.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, 1007–1012. AAAI Press.

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

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, 161–170. AAAI Press.

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

Holte, R. C.; Felner, A.; Newton, J.; Meshulam, R.; and Furcy, D. 2006. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence* 170(16–17):1123–1136.

Karpas, E., and Domshlak, C. 2009. Cost-optimal planning with landmarks. In Boutilier, C., ed., *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, 1728–1733. AAAI Press.

	h^{SCP} over different PDB heuristics					IPC 2018 Planners			Scorpion			
Coverage	HC	SYS-3-LIM	CPC	CEGAR	SYS-SCP	Comp1	Comp2	Delfi1	COMB	SYS-SCP	SYS-SCP +CART	SYS-SCP +COMB
agricola (20)	0	0	0	2	0	6	5	**13**	2	3	3	4
airport (50)	36	23	33	36	38	21	24	27	39	**46**	**46**	**46**
barman (34)	4	4	11	4	11	11	11	**16**	4	11	11	11
blocks (35)	28	28	28	28	28	30	30	**32**	28	28	28	28
childsnack (20)	0	0	0	0	0	0	0	**6**	0	0	0	0
data-network (20)	12	11	13	13	13	12	13	13	**14**	13	**14**	**14**
depot (22)	11	12	11	13	13	7	7	11	14	14	**15**	**15**
driverlog (20)	13	**15**	13	**15**	**15**	13	14	14	**15**	**15**	**15**	**15**
elevators (50)	41	43	43	40	41	37	37	**44**	**44**	41	**44**	**44**
floortile (40)	6	11	15	11	16	**34**	**34**	29	16	33	32	**34**
freecell (80)	21	26	25	27	30	22	26	26	**70**	31	68	65
ged (20)	**19**	**19**	**19**	**19**	**19**	16	**19**	15	**19**	**19**	**19**	**19**
grid (5)	**3**	**3**	**3**	**3**	**3**	2	2	**3**	**3**	**3**	**3**	**3**
gripper (20)	8	8	7	8	8	**20**	**20**	**20**	8	8	8	8
hiking (20)	13	14	**19**	18	15	15	18	18	14	15	15	14
logistics (63)	30	33	30	34	**36**	26	28	28	35	**36**	**36**	35
miconic (150)	72	143	116	144	144	105	104	142	**145**	**145**	144	**145**
movie (30)	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**
mprime (35)	24	**34**	23	31	**34**	19	21	23	30	**34**	**34**	**34**
mystery (30)	17	**19**	16	**19**	**19**	13	16	17	**19**	**19**	**19**	**19**
nomystery (20)	**20**	**20**	**20**	**20**	**20**	12	19	18	**20**	**20**	**20**	**20**
openstacks (100)	49	53	49	49	53	74	73	**89**	51	53	53	53
organic (20)	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**	**7**
organic-split (20)	10	10	10	10	10	13	13	**14**	13	13	13	13
parcprinter (50)	40	32	34	32	41	38	41	48	**50**	**50**	**50**	**50**
parking (40)	**13**	8	9	12	**13**	2	2	**13**	**13**	**13**	**13**	**13**
pathways (30)	4	4	4	**5**	**5**	**5**	4	**5**	**5**	**5**	**5**	**5**
pegsol (50)	**50**	48	48	48	48	48	48	48	**50**	48	48	**50**
petri-net (20)	0	7	4	4	8	16	18	**20**	0	7	6	0
pipes-nt (50)	21	24	**25**	23	23	15	22	**25**	**25**	23	**25**	**25**
pipes-t (50)	18	18	17	17	18	13	16	**22**	18	18	18	18
psr-small (50)	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**	**50**
rovers (40)	8	8	10	9	10	**14**	13	**14**	11	12	13	13
satellite (36)	6	6	7	8	9	11	10	**14**	9	10	10	10
scanalyzer (50)	23	37	25	31	**41**	21	21	34	33	**41**	**41**	**41**
snake (20)	**14**	12	13	12	13	11	13	11	13	13	13	13
sokoban (50)	**50**	**50**	**50**	**50**	**50**	46	48	**50**	**50**	**50**	**50**	**50**
spider (20)	15	14	14	14	14	10	11	11	**16**	14	15	**16**
storage (30)	16	16	16	16	16	15	15	**19**	16	16	16	16
termes (20)	12	12	13	12	13	14	**15**	10	13	13	12	12
tetris (17)	10	11	11	10	11	10	**13**	**13**	**13**	**13**	**13**	**13**
tidybot (40)	22	28	24	26	30	24	29	29	**32**	30	**32**	**32**
tpp (30)	6	7	12	12	12	9	**14**	10	8	12	12	12
transport (70)	35	34	36	34	36	27	29	31	35	36	**37**	**37**
trucks (30)	9	9	10	13	12	12	10	12	12	13	**16**	**16**
visitall (40)	28	**30**	**30**	**30**	**30**	16	21	**30**	**30**	**30**	**30**	**30**
woodwork (50)	29	44	39	36	49	45	46	**50**	**50**	**50**	**50**	**50**
zenotravel (20)	**13**	**13**	**13**	**13**	**13**	**13**	**13**	12	**13**	**13**	**13**	**13**
Sum (1827)	966	1088	1055	1098	1168	1030	1093	1236	1205	1217	**1265**	1261

Table 6: Number of tasks solved by different planners.

Katz, M., and Domshlak, C. 2008. Optimal additive composition of abstraction-based admissible heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 174–181. AAAI Press.

Katz, M., and Domshlak, C. 2010. Optimal admissible composition of abstraction heuristics. *Artificial Intelligence* 174(12–13):767–798.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artificial Intelligence* 134(1–2):9–22.

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening A*. *Artificial Intelligence* 129:199–218.

Korf, R. E. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997)*, 700–705. AAAI Press.

Lelis, L. H. S.; Stern, R.; and Sturtevant, N. R. 2014. Estimating search tree size with duplicate detection. In Edelkamp, S., and Barták, R., eds., *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS 2014)*, 114–122. AAAI Press.

Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pommerening, F., and Seipp, J. 2016. Fast Downward dead-end pattern database. In Muise, C., and Lipovetzky, N., eds., *Unsolvability International Planning Competition: planner abstracts*, 2.

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. 2015. From non-negative to general operator cost partitioning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI 2015)*, 3335–3341. AAAI Press.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the most out of pattern databases for classical planning. In Rossi, F., ed., *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2357–2364. AAAI Press.

Pommerening, F. 2017. *New Perspectives on Cost Partitioning for Optimal Classical Planning*. Ph.D. Dissertation, University of Basel.

Rovner, A.; Sievers, S.; and Helmert, M. 2019. Counterexample-guided abstraction refinement for pattern selection in optimal classical planning. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*. AAAI Press.

Seipp, J., and Helmert, M. 2014. Diverse and additive Cartesian abstraction heuristics. In Chien, S.; Fern, A.; Ruml, W.; and Do, M., eds., *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 289–297. AAAI Press.

Seipp, J., and Helmert, M. 2018. Counterexample-guided Cartesian abstraction refinement for classical planning. *Journal of Artificial Intelligence Research* 62:535–577.

Seipp, J., and Helmert, M. 2019. Subset-saturated cost partitioning for optimal classical planning. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*. AAAI Press.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Seipp, J.; Keller, T.; and Helmert, M. 2017a. A comparison of cost partitioning algorithms for optimal classical planning. In Barbulescu, L.; Frank, J.; Mausam; and Smith, S. F., eds., *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, 259–268. AAAI Press.

Seipp, J.; Keller, T.; and Helmert, M. 2017b. Narrowing the gap between saturated and optimal cost partitioning for classical planning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI 2017)*, 3651–3657. AAAI Press.

Seipp, J. 2017. Better orders for saturated cost partitioning in optimal classical planning. In Fukunaga, A., and Kishimoto, A., eds., *Proceedings of the 10th Annual Symposium on Combinatorial Search (SoCS 2017)*, 149–153. AAAI Press.

Seipp, J. 2018. Fast Downward Scorpion. In *Ninth International Planning Competition (IPC-9): planner abstracts*, 77–79.

Sievers, S.; Katz, M.; Sohrabi, S.; Samulowitz, H.; and Ferber, P. 2019. Deep learning for cost-optimal planning: Task-dependent planner selection. In *Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence (AAAI 2019)*. AAAI Press.

Torralba, Á.; Linares López, C.; and Borrajo, D. 2018. Symbolic perimeter abstraction heuristics for cost-optimal planning. *Artificial Intelligence* 259:1–31.

Yang, F.; Culberson, J.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research* 32:631–662.

A* Search and Bound-Sensitive Heuristics for Oversubscription Planning

Michael Katz[1] , **Emil Keyder**[2]
[1]IBM Research, Yorktown Heights, NY, USA
[2]Invitae Corporation, San Francisco, CA, USA
michael.katz1@ibm.com, emilkeyder@gmail.com

Abstract

Oversubscription planning (OSP) is the problem of finding plans that maximize the utility value of their end state while staying within a specified cost bound. Recently, it has been shown that OSP problems can be reformulated as classical planning problems with multiple cost functions but no utilities. Here we take advantage of this reformulation to show that OSP problems can be solved optimally using the A^* search algorithm, in contrast to previous approaches that have used variations on branch-and-bound search. This allows many powerful techniques developed for classical planning to be applied to OSP problems. We also introduce novel bound-sensitive heuristics, which are able to reason about the primary cost of a solution while taking into account secondary cost functions and bounds, to provide superior guidance compared to heuristics that do not take these bounds into account. We implement two such bound-sensitive variants of existing classical planning heuristics, and show experimentally that the resulting search is significantly more informed than comparable heuristics that do not consider bounds.

Introduction

Oversubscription planning (OSP) problems are a family of deterministic planning problems. In contrast to classical planning, where a set of hard goals is specified and the planner searches for a minimal (or low) cost plan that reaches a state in which all of the goals are made true, oversubscription planning specifies a *utility function* that describes the benefit associated with achieving different possible states, and asks for a plan whose cost does not exceed a set bound and achieves as high a utility as possible [Smith, 2004].

While domain-independent classical planning approaches have increasingly standardized around variations on A^* search and heuristics that are automatically extracted from the problem description [Bonet and Geffner, 2001; Keyder and Geffner, 2008; Haslum and Geffner, 2000; Edelkamp, 2001; Helmert *et al.*, 2014; Helmert and Domshlak, 2009], OSP has generally been solved with branch-and-bound algorithms and heuristics that compute an admissible (in this context non-under) estimate of the utility achievable from a state. In order to obtain these estimates, recent approaches often adapt classical planning techniques such as landmarks [Mirkis and Domshlak, 2014; Muller and Karpas, 2018] or abstractions [Mirkis and Domshlak, 2013], and enhance them with reasoning that is specific to the context of OSP, such as the knowledge that there always exists an optimal plan that ends with a utility-increasing action, or that the cost bound for the problem can be reduced under specific conditions to aid the search algorithm in detecting that improving over the currently achieved utility is impossible.

In contrast to these approaches, our aim here is to show that general methods from classical planning, including A^* search, can be used in the OSP setting nearly as is. This previously turned out to be the case for the related *net-benefit* planning problem, where classical planners solving a compilation were shown to outperform planners designed specifically for that task [Keyder and Geffner, 2009]. Here, we use a similar, recently proposed compilation that converts OSP problems into classical planning problems with multiple cost functions but no utilities [Katz *et al.*, 2019a]. In addition, we demonstrate that existing classical planning heuristics can be used to guide the search for optimal plans. While these heuristics are typically uninformative out-of-the-box, they require only minor modifications (and no specific reasoning about utilities) to render them sensitive to the secondary cost functions and bounds that are introduced by the compilation. Our experiments with A^* and the newly introduced estimators that we refer to as *bound-sensitive heuristics* show that they lead to informed searches that are competitive with, and in some cases outperform, the state of the art for optimal OSP.

One related area of research in the classical setting is that of *bounded-cost planning*, where the planner looks for *any* plan with (primary) cost below a given bound, similar to the treatment of the secondary cost in the OSP setting. Approaches proposed for this setting include dedicated search algorithms [Stern *et al.*, 2011] and heuristics that take into account accumulated cost and plan length at the current search node [Thayer and Ruml, 2011; Haslum, 2013; Dobson and Haslum, 2017]. These approaches work by preferentially expanding nodes in areas of the search space that are likely to have a solution under the cost bound. Optimal OSP, however, requires expanding all nodes that potentially

lie on a path to state with maximal utility. Furthermore, it cannot be assumed that solutions necessarily achieve all soft goals. Heuristics that are able to take into account bounds on secondary cost functions have also been investigated in the stochastic shortest path setting, where they were used as additional constraints in an LP-based heuristic to consider limitations on fuel or time resources [Trevizan *et al.*, 2017].

We now briefly review the various flavors of planning that we consider in this work, and introduce the formalisms by which we describe them.

Background

We describe planning problems in terms of extensions to the SAS$^+$ formalism [Bäckström and Nebel, 1995]. A *classical planning task* $\Pi = \langle V, O; s_I, G, \mathcal{C} \rangle$ is given by a set of variables V, with each variable $v \in V$ having a finite domain $dom(v)$, a set of actions O, with each action $o \in O$ described by a pair $\langle \mathsf{pre}(o), \mathsf{eff}(o) \rangle$ of partial assignments to V, called the *precondition* and *effect* of o, respectively, initial state s_I and goal condition G, which are full and partial assignments to V, respectively, and the cost function $\mathcal{C} : O \to \mathbb{R}^{0+}$. A state s is given by a full assignment to V. An action is said to be *applicable* in a state s if $\mathsf{pre}(o) \subseteq s$, and $s[\![o]\!]$ denotes the result of applying o in s, where the value of each $v \in V$ is given by $\mathsf{eff}(o)[v]$ if defined and $s[v]$ otherwise. An operator sequence $\pi = \langle o_1, \ldots, o_k \rangle$ is applicable in s if there exist states $s_0, \cdots, s_k$ such that (i) $s_0 = s$, and (ii) for each $1 \leq i \leq k$, o_i is applicable in s_{i-1} and $s_i = s_{i-1}[\![o_i]\!]$. We refer to the state s_k by $s[\![\pi]\!]$ and call it the end state of π. An operator sequence π is a plan for a classical planning problem if it is applicable in s_I and $G \subseteq s_I[\![\pi]\!]$. The cost of a plan π is given by $\mathcal{C}(\pi) = \sum_{o \in \pi} \mathcal{C}(o)$; the goal of optimal classical planning is to find a plan with minimal cost. We refer to a pair of variable v and its value $\vartheta \in dom(v)$ as a *fact* and denote it by $\langle v, \vartheta \rangle$. We sometimes abuse notation and treat partial assignments as sets of facts.

An oversubscription planning (OSP) problem is given by $\Pi_{\text{OSP}} = \langle V, O, s_I, \mathcal{C}, \mathsf{u}, \mathsf{B} \rangle$, where V, O, s_I, and $\mathcal{C}$ are as in classical planning, $\mathsf{u} : (\langle v, \vartheta \rangle) \to \mathbb{R}^{0+}$ is a non-negative valued utility function over variable assignments (facts), and B is a cost bound for the plan, imposing the additional requirement that only plans π such that $\mathcal{C}(\pi) \leq \mathsf{B}$ are valid. The *utility* of a plan π is given by $\sum_{\langle v, \vartheta \rangle \in s_I[\![\pi]\!]} \mathsf{u}(\langle v, \vartheta \rangle)$; the objective of OSP problems is to find valid plans with maximal utility.

A multiple cost function (MCF) problem is given by $\Pi_{\text{MCF}} = \langle V, O, s_I, G, \mathcal{C}_0, \mathscr{C} \rangle$, where V, O, s_I, and $\mathcal{C}_0$ are as in classical planning, $\mathcal{C}_0$ is the *primary cost function*, and $\mathscr{C} = \{\langle \mathcal{C}_i, \mathsf{B}_i \rangle \mid 1 \leq i \leq n\}$ is a set of *secondary cost functions* $\mathcal{C}_i : O \to \mathbb{R}^{0+}$, and *bounds*, both non-negative. Valid plans for MCF planning problems fulfill the condition $\mathcal{C}_i(\pi) \leq \mathsf{B}_i$ for all secondary cost functions, and optimal plans for MCF planning have minimal primary cost $\mathcal{C}_0(\pi)$. In this paper we only consider MCF problems with a single secondary cost function, i.e. $n = 1$.

Reformulating OSP Problems

It has recently been shown that an OSP problem can be compiled into an MCF planning problem with a single secondary cost function that corresponds to the cost function $\mathcal{C}$ of the original problem, and is constrained to not exceed the specified bound B [Katz *et al.*, 2019a]. The primary cost function for the problem, or the cost function to be optimized, results from compiling the utilities from the original problem into costs. Two different compilations have been proposed for this task: (i) the *soft goals compilation*, which adds for each variable v that has some value $\vartheta \in dom(v)$ for which a utility is specified, a hard goal, along with actions that are able to achieve this hard goal at different costs, and (ii) the *state delta compilation* which encodes in the cost of each action the change in state utility that results from applying it. Here we consider only (i), as (ii) introduces negative action costs that A^* and existing classical planning heuristics are not designed to handle. Note, however, that our methods do not depend on the specific choice of compilation, as long as they remove utilities from the problem and do not introduce negative action costs.

The *soft goals compilation* was originally introduced in the context of net-benefit planning, which is similar to oversubscription planning but does not specify a bound on plan cost, having instead as an objective the minimization of the difference between the achieved utility and the cost of the plan [Keyder and Geffner, 2009]. It can be applied in the OSP setting to result in an MCF planning problem as follows:

Definition 1 *Let* $\Pi_{\text{OSP}} = \langle V, O, s_I, \mathcal{C}, \mathsf{u}, \mathsf{B} \rangle$ *be an oversubscription planning task. The* soft goals reformulation $\Pi^{sg}_{\text{MCF}} = \langle V', O', s_I, G', \mathcal{C}_0, \{\langle \mathcal{C}', \mathsf{B} \rangle\} \rangle$ *of* Π_{OSP} *is an* MCF planning task*, where*

- $V' = \{v' | v \in V\}$*, with* $dom(v') = \begin{cases} dom(v) \cup \{g_v\} & \mathsf{u}_{max}(v) > 0 \\ dom(v) & \textit{otherwise}, \end{cases}$
- $O' = O \cup \{o^{v,\vartheta} = \langle \{\langle v, \vartheta \rangle\}, \{\langle v, g_v \rangle\} \rangle \mid \vartheta \in dom(v), v \in V, \mathsf{u}_{max}(v) > 0\}$
- $G' = \{\langle v, g_v \rangle | v \in V, \mathsf{u}_{max}(v) > 0\}$,
- $\mathcal{C}_0(o) = \begin{cases} 0 & o \in O \\ \mathsf{u}_{max}(v) - \mathsf{u}(\langle v, \vartheta \rangle) & o = o^{v,\vartheta}, \end{cases}$
- $\mathcal{C}'(o) = \begin{cases} \mathcal{C}(o) & o \in O \\ 0 & \textit{otherwise}, \end{cases}$

with $\mathsf{u}_{max}(v) := \max_{\vartheta \in dom(v)} \mathsf{u}(\langle v, \vartheta \rangle)$ *denoting the maximum utility over the values of the variable* v*.*

In the reformulated problem, only the $o^{v,\vartheta}$ actions for which ϑ is not the maximum utility value of v have positive primary costs. These actions make explicit that a particular utility will not be achieved, and that the plan has instead chosen to achieve the associated g_v by accepting the associated cost penalty. The *primary cost* of a plan π for the reformulated problem is then given by $\sum_{v \in V} \mathsf{u}_{max}(v) - \sum_{f \in s[\![\pi]\!]} \mathsf{u}(f)$.

Note that this compilation assumes that utilities are defined for single facts. The more general case, in which utilities are instead defined for logical formulae φ, can be handled as in the soft goals compilation by introducing a new variable v_φ, and two actions that achieve its goal value with cost 0 and precondition φ, and cost $\mathsf{u}(\varphi)$ and precondition $\emptyset$, respectively [Keyder and Geffner, 2009]. Since we consider only single fact utilities here, we do not discuss this case in detail.

While this compilation is sound as stated, two further optimizations can be made to reduce the state space of the resulting compiled problem. First, an arbitrary ordering can be introduced over V to ensure that the g_v values are achieved in a fixed sequence, to avoid searching over different orderings. Second, a new precondition fact that is deleted by the $o^{v,\vartheta}$ actions can be added to the original domain actions to ensure that $o^{v,\vartheta}$ actions happen only at the end of the plan and are not interleaved with the original domain actions. We make use of both of these optimizations here.

A^* for MCF Planning Problems

The A^* algorithm extends blind search techniques such as Dijkstra's algorithm by allowing the incorporation of admissible (non-overestimating) heuristics [Hart *et al.*, 1968]. In each iteration of its main loop, A^* picks a node n to expand with minimal $f(n) = g(n) + h(n)$ value, where $g(n)$ is the cost of the path to n, and $h(n)$ is an admissible estimate of the remaining cost to the goal. An optimal solution to the problem is found when a node n with minimal $f(n)$ value is a goal node.

To adapt A^* to the MCF planning setting, we store at each node n a set of accumulated path costs $g_i(n)$ resulting from each of the secondary cost functions $\mathcal{C}_1, \ldots, \mathcal{C}_n$, in addition to the accumulated primary cost $g_0(n)$. When a node is taken from the priority queue and expanded, generated successor nodes for which any $g_i(n) > \mathsf{B}_i$ can be immediately pruned, as all $\mathcal{C}_i$ are assumed to be non-negative, and they cannot constitute valid prefixes for solution paths.

One key optimization used in modern A^* implementations in the classical setting is duplicate detection, which allows states that are rediscovered during search to be discarded, if the new g value exceeds the cost of the path to the state that was previously found, or to be updated with a new parent, if the cost of the new path is less. In the MCF setting, care must be taken to ensure that newly discovered nodes are discarded (or replace existing nodes), only when they are dominated by (or dominate), the existing node in all cost dimensions. While the only necessary property of the open list from a correctness perspective is that it order nodes by increasing primary $f(n)$ value, the choice of a secondary ordering heuristic plays a role here: an ordering that causes a dominating node to be generated first and enables subsequently generated nodes to be immediately discarded as dominated results in superior performance. In our implementation of the algorithm, we therefore use an open list that orders nodes by increasing $g_i(n)$ value when their primary $f(n)$ values are the same.

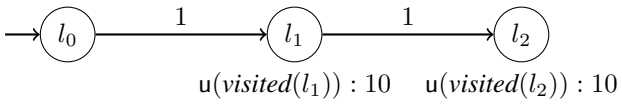


Figure 1: An OSP problem based on the VISIT-ALL domain.

Bound-Sensitive Heuristics

While any admissible heuristic can be used to guide search in MCF planning, classical planning heuristics that ignore bounds entirely are typically extremely uninformative. Consider the problem shown in Figure 1: the agent is initially at l_0, and can obtain a utility of 10 by visiting each of the locations l_1 and l_2. The costs of the actions $\mathit{move}(l_0, l_1)$ and $\mathit{move}(l_1, l_2)$ are both 1. In the compiled MCF version of this problem, an optimal but naive heuristic that ignores the bound will give an estimate for the primary cost of 0, as both $\mathit{visited}(l_1)$ and $\mathit{visited}(l_2)$ can be made true, and the associated 0-primary cost $o^{\mathit{visited}(l_*)}$ actions applied to reach the newly introduced hard goals corresponding to each utility. If, however, $\mathsf{B} = 1$, the optimal $\mathcal{C}_0$ cost at l_0 is 10, since l_2 cannot be reached at cost $\leq \mathsf{B}$ and the agent must use the $o^{\mathit{not\text{-}visited}(l_2)}$ action to achieve the associated hard goal with a cost of 10. Similarly, if $\mathsf{B} = 0$, the $\mathcal{C}_0$ cost of the optimal plan is 20, since the value of $\mathcal{C}_1$ for all available actions exceeds the bound B. In practice, it turns out that the OSP versions of many classical planning problems have similar behavior: their state spaces are strongly connected, so any variable assignment can be achieved from any state, and classical planning heuristics that ignore bounds are no more informed than blind search.

In order to obtain estimates that take secondary cost bounds into account and can guide heuristic search towards feasible solutions, we therefore introduce *bound-sensitive heuristics*. In the following, we use $\mathbf{b}$ to denote a *budget vector* of non-negative reals that indicate the unused component of each of the secondary cost bounds B_i at a given search node.

Definition 2 (Optimal bound-sensitive heuristic) *Given an MCF planning problem $\Pi_{\mathrm{MCF}} = \langle V, O, s_I, G, \mathcal{C}_0, \mathscr{C} \rangle$, the* optimal bound-sensitive heuristic *$h^*(s, \boldsymbol{b})$ for a state s and budget vector $\boldsymbol{b}$ is given by the minimal primary cost $\mathcal{C}_0(\pi)$ of a plan π for s such that $\mathcal{C}_i(\pi) \leq \boldsymbol{b}_i$ for $i = 1, \ldots, n$.*

By analogy with standard admissible heuristics, an *admissible bound-sensitive heuristic* is a non-overestimating bound-sensitive heuristic:

Definition 3 (Admissible bound-sensitive heuristic) *Given an MCF planning problem $\Pi_{\mathrm{MCF}} = \langle V, O, s_I, G, \mathcal{C}_0, \mathscr{C} \rangle$, an* admissible bound-sensitive heuristic *$h(s, \boldsymbol{b})$ for a state s and budget vector $\boldsymbol{b}$ is a heuristic h such that $h(s, \boldsymbol{b}) \leq h^*(s, \boldsymbol{b})$ for all s, $\boldsymbol{b}$.*

Any classical planning heuristic that completely ignores $\mathcal{C}_i$ and B_i can be thought of as an admissible bound-sensitive heuristic that assumes $\mathbf{b} = \infty$. As the value of $\mathbf{b}$ decreases, the value of $h^*(s, \mathbf{b})$ can only increase. In general, it is useful to keep in mind the following property:

Theorem 1 *Given a state s and budget vectors $\boldsymbol{b}, \boldsymbol{b'}$ such that $\boldsymbol{b} \leq \boldsymbol{b'}$ (where $\leq$ is interpreted as a pairwise comparison), $h^*(s, \boldsymbol{b}) \geq h^*(s, \boldsymbol{b'})$.*

Proof sketch: This follows from the fact that any plan π for s such that $\mathcal{C}_i(\pi) \leq \mathbf{b}_i$ also has the property that $\mathcal{C}_i(\pi) \leq \mathbf{b'}_i$ for $i = 1, \ldots, n$ since $\mathbf{b} \leq \mathbf{b'}$, yet the opposite is not the case. ∎

Theorem 1 applied to MCF planning problems obtained as the *soft goals compilations* of OSP problems states that for any s, decreasing $\mathbf{b}$ *increases* $h^*(s, \mathbf{b})$, and decreases the achievable utility, since the primary cost here indicates the utility that the plan must declare unachievable through $o^{v,\vartheta}$ actions with $\mathcal{C}_0(o^{v,\vartheta}) \geq 0$.

Bound-Sensitive h^{max}

The admissible classical heuristic h^{max} estimates the cost of a set of facts F as the cost of the most expensive fact $f \in F$, and applies this approximation recursively to action preconditions in order to obtain the cost of the goal [Bonet and Geffner, 2001]:

$$h^{max}_{\mathcal{C}}(F, s) = \max_{f \in F} h^{max}_{\mathcal{C}}(f, s)$$

$$h^{max}_{\mathcal{C}}(f, s) = \begin{cases} 0 & f \in s \\ \min_{o \in achievers(f,s)} h^{max}_{\mathcal{C}}(o, s) & otherwise \end{cases}$$

$$h^{max}_{\mathcal{C}}(o, s) = \mathcal{C}(o) + h^{max}_{\mathcal{C}}(\mathsf{pre}(o), s)$$

where $h^{max}_{\mathcal{C}}$ denotes the value of h^{max} computed with a cost function $\mathcal{C}$, and $achievers(f, s)$ denotes the set of actions o for which $f \in \mathsf{eff}(o)$. Note that the h^{max} cost of a fact f that is not present in s is computed by choosing an action o from this set that achieves it with minimum possible cost. Given a set of secondary cost functions and bounds $\mathscr{C} = \{\langle \mathcal{C}_1, \mathsf{B}_1 \rangle, \ldots, \langle \mathcal{C}_n, \mathsf{B}_n \rangle\}$, a bound-sensitive version of h^{max} can easily be obtained by replacing the set of achievers used to compute $h^{max}_{\mathcal{C}_0}$ with

$$achievers(f, s)_{\mathcal{C}_0} = \{o \mid f \in \mathsf{eff}(o) \wedge \bigwedge_{i=1,\ldots,n} h^{max}_{\mathcal{C}_i}(o, s) \leq \mathsf{B}_i\}$$

where actions o for which any estimate $h^{max}_{\mathcal{C}_i}(o, s)$ exceeds B_i are not considered. Note that due to the admissibility of h^{max}, this restriction of the set of achievers is sound but not complete: it is guaranteed that any action removed from the set of achievers cannot be used in a valid plan, but there may be additional actions that cannot be achievers but are not pruned by the heuristic. In general, any admissible estimate $h^{max}_{\mathcal{C}_i}(o, s)$ could be used to compute $achievers(f, s)_{\mathcal{C}_0}$, but we have chosen h^{max} here for simplicity.

Theorem 2 *Bound-sensitive $h^{max}_{\mathcal{C}_0}$ is an admissible bound-sensitive heuristic.*

Proof sketch: This follows from the admissibility of the heuristic used to compute $achievers(f, s)_{\mathcal{C}_0}$. ∎

Bound-Sensitive Merge-and-shrink

Merge-and-shrink heuristics are a family of abstraction heuristics that incrementally build a representation of the full state space of a problem [Helmert *et al.*, 2014]. The construction process begins with the set of transition systems induced over each state variable; at each step, two transition systems are selected to be merged and replaced with their synchronized product. Since the transition systems need to be represented explicitly in memory, before the merge a shrinking step is perfomed on the two selected transition systems to enforce a user-specified threshold on the size of the synchronized product. This step is performed by abstracting multiple states in the current representation into a single state (and thereby losing optimality). The final output of the algorithm consists of a single abstract transition system in which multiple states and actions from the original task are mapped to a single state or transition, respectively. $h^{\text{MS}}(s)$ is then given by the cost of a shortest path from the abstract state representing s to the closest abstract goal state in the final transition system. This estimate is admissible by definition.

To adapt *merge-and-shrink* to the MCF setting, we maintain for each transition in the abstract state space the minimum $\mathcal{C}_i$ cost for $i = 1, \ldots, n$ among all of the transitions from the original task represented by that transition. The distance $\mathcal{C}_i$ between any two abstract states s, s' then represents a non-overestimate of the secondary cost of reaching s' from s. A bound-sensitive heuristic value for a state s can be computed as the minimum $\mathcal{C}_0$ cost of a path π from s to an abstract goal state s_g whose $\mathcal{C}_i$ cost in the abstract state space does not exceed B_i, for any i. The $\mathcal{C}_0$ cost of such such a path can be computed with a modified version of Dijkstra's algorithm that stores secondary cost information for each node and discards nodes for which $\mathcal{C}_i > \mathsf{B}_i$ for any i.

Theorem 3 *Bound-sensitive h^{MS} is an admissible bound-sensitive heuristic.*

Proof sketch: This follows from the fact that the secondary costs used in the abstract state space are the minimums of the secondary costs $\mathcal{C}_i$ of the represented transitions in the original problem, and the proof of admissibility of standard h^{MS}. ∎

While the ms^b heuristic can be implemented by running Dijkstra's algorithm in the abstract state space for each heuristic computation, an important optimization when a single secondary cost function is present (which is the case in the compiled OSP problems that we consider) is to run Dijkstra only once during preprocessing, and compute the primary cost in the presence of different bounds on the secondary cost. This information can then be stored as a sequence of pairs $\langle \langle b_0, c_0 \rangle, \ldots, \langle b_n, c_n \rangle \rangle$, where $b_0, \ldots, b_n$ is strictly increasing and $c_0, \ldots, c_n$ is strictly decreasing (recall Theorem 1). $h^{\text{MS}}(s, \mathbf{b})$ is then given by the first c_i such that $\mathbf{b}_i \leq \mathbf{b}$.

Experiments

We implemented our approach in the Fast Downward planner [Helmert, 2006], and evaluated it on a set of publically

	25						50						75						100					
Coverage	BnB	bl	max^b	max	ms^b	ms	BnB	bl	max^b	max	ms^b	ms	BnB	bl	max^b	max	ms^b	ms	BnB	bl	max^b	max	ms^b	ms
airport	**27**	±0	-1	-1	-9	-9	**22**	±0	±0	-1	-4	-4	**21**	±0	-1	±0	-4	-4	**21**	±0	-3	-3	-5	-5
barman11	12	±0	**+1**	±0	±0	±0	**8**	±0	±0	±0	±0	±0	**4**	±0	±0	±0	±0	±0	**4**	±0	±0	±0	±0	±0
barman14	6	±0	±0	±0	**+2**	±0	**3**	±0	±0	-3	±0	±0	**0**	±0	±0	±0	±0	±0	**0**	±0	±0	±0	±0	±0
blocks	**35**	±0	±0	±0	±0	±0	28	±0	+1	-2	**+4**	±0	21	±0	±0	±0	**+8**	±0	18	±0	±0	±0	**+8**	±0
childsnack14	0	±0	+1	±0	**+2**	±0	**0**	±0	±0	±0	±0	±0	**0**	±0	±0	±0	±0	±0	**0**	±0	±0	±0	±0	±0
depot	**16**	±0	-1	-2	-1	±0	**11**	±0	±0	-4	±0	-1	**7**	±0	±0	-1	±0	±0	4	±0	±0	±0	**+1**	±0
driverlog	**15**	±0	±0	±0	±0	±0	13	±0	**+1**	-1	**+1**	±0	10	±0	+1	±0	**+2**	±0	7	±0	+1	±0	**+4**	±0
elevators08	**30**	±0	±0	-1	-1	±0	**25**	±0	-1	-1	±0	±0	23	±0	-1	-1	**+1**	±0	17	+1	±0	-1	**+3**	+1
elevators11	**20**	±0	±0	±0	±0	±0	**19**	±0	±0	±0	±0	±0	18	±0	-1	-1	**+1**	±0	14	+1	±0	-1	**+2**	+1
floortile11	**9**	±0	±0	±0	-2	±0	4	±0	**+1**	±0	±0	±0	2	±0	**+2**	**+2**	+1	±0	2	±0	**+4**	**+4**	±0	±0
floortile14	**9**	±0	±0	±0	-2	-3	**2**	±0	±0	±0	±0	±0	0	±0	**+2**	+1	±0	±0	0	±0	**+5**	**+5**	±0	±0
freecell	**77**	±0	-14	-33	-12	-6	**30**	±0	-2	-13	-2	-1	**21**	±0	-6	-6	-1	-1	**20**	±0	-6	-6	-2	-4
ged14	**20**	±0	±0	±0	±0	±0	**20**	±0	±0	±0	±0	±0	**20**	±0	±0	±0	-1	±0	**20**	±0	±0	±0	±0	±0
grid	**5**	±0	±0	-1	±0	±0	**3**	±0	±0	±0	-1	±0	**2**	±0	±0	±0	-1	±0	1	±0	±0	±0	**+1**	±0
gripper	11	±0	±0	±0	**+1**	±0	**8**	±0	±0	±0	±0	±0	**8**	±0	-1	±0	±0	±0	**8**	±0	-1	±0	±0	±0
hiking14	19	±0	-1	-5	**+1**	±0	14	±0	-1	-3	**+3**	±0	13	±0	-2	-2	**+2**	±0	11	±0	-2	-2	**+3**	±0
logistics00	21	±0	**+1**	±0	±0	±0	**16**	±0	±0	±0	±0	±0	12	±0	**+2**	±0	**+2**	±0	10	±0	±0	±0	**+4**	±0
logistics98	6	±0	**+1**	±0	±0	±0	4	±0	**+1**	±0	**+1**	±0	2	±0	**+1**	±0	**+1**	±0	**2**	±0	±0	±0	±0	±0
miconic	96	±0	-1	-4	**+12**	-1	65	±0	±0	-1	**+7**	±0	55	±0	±0	±0	**+11**	±0	50	+5	±0	±0	**+11**	+4
mprime	**35**	±0	±0	-2	-4	-2	**28**	-1	-1	-5	-3	-1	**24**	±0	-1	-2	-2	±0	19	±0	**+1**	-5	-2	±0
mystery	**29**	±0	±0	±0	-2	±0	**27**	-1	±0	-3	-4	-1	**21**	±0	±0	-3	-1	±0	**18**	±0	±0	-3	-1	±0
nomystery11	**20**	±0	±0	±0	±0	±0	**14**	±0	±0	-2	±0	±0	**10**	±0	-1	-2	±0	±0	8	±0	±0	±0	**+3**	+1
openstacks08	**30**	±0	±0	±0	±0	±0	**25**	±0	±0	±0	±0	±0	**24**	±0	±0	±0	±0	±0	**22**	±0	-3	-2	±0	±0
openstacks11	**20**	±0	±0	±0	±0	±0	**18**	±0	±0	±0	±0	±0	**17**	±0	±0	±0	±0	±0	**17**	±0	-3	-3	±0	±0
openstacks14	**20**	-1	-1	-1	-1	-1	**15**	-2	-4	-4	-2	-2	**7**	±0	-3	-3	±0	±0	**3**	±0	-1	±0	±0	±0
openstacks	**9**	±0	-2	-2	-2	-2	**7**	±0	±0	±0	±0	±0	**7**	±0	±0	±0	±0	±0	**7**	±0	±0	±0	±0	±0
parcprinter08	17	-2	**+1**	-2	-3	-3	13	±0	**+1**	±0	±0	-1	11	±0	**+2**	±0	±0	-1	11	-1	**+2**	**+2**	+1	±0
parcprinter11	13	-1	**+1**	-2	-2	-2	9	±0	**+1**	±0	±0	±0	7	±0	**+2**	±0	+1	±0	6	±0	**+3**	+2	+2	+2
parking11	**11**	-1	-1	-2	-3	-1	**1**	±0	±0	±0	±0	±0	**0**	±0	±0	±0	±0	±0	**0**	±0	±0	±0	±0	±0
parking14	**14**	-2	-3	-6	-3	-3	**4**	±0	-3	-4	±0	±0	0	±0	±0	±0	**+1**	±0	**0**	±0	±0	±0	±0	±0
pathways-nn	**5**	±0	±0	±0	±0	±0	4	±0	**+1**	±0	**+1**	±0	**4**	±0	±0	±0	±0	±0	**4**	±0	±0	±0	±0	±0
pegsol08	**30**	±0	±0	±0	±0	±0	**30**	±0	±0	±0	±0	±0	**29**	-1	±0	-2	±0	±0	**27**	±0	±0	±0	±0	±0
pegsol11	**20**	±0	±0	±0	±0	±0	**20**	±0	±0	±0	±0	±0	**19**	-2	±0	-2	±0	±0	**17**	±0	±0	±0	±0	±0
pipes-notank	**45**	±0	±0	-2	-30	-27	**30**	±0	-1	-5	-14	-12	**22**	±0	-2	-6	-5	-5	**15**	±0	-1	-2	±0	-1
pipes-tank	**35**	-2	-6	-11	-9	-9	**20**	±0	-3	-5	-3	-3	**16**	-1	-4	-5	-1	-1	**11**	±0	-1	-3	±0	±0
psr-small	**50**	±0	±0	±0	±0	±0	**50**	±0	±0	±0	±0	±0	49	±0	±0	±0	**+1**	±0	**49**	±0	±0	±0	±0	±0
rovers	15	±0	**+1**	-2	-1	±0	8	±0	**+1**	±0	**+1**	±0	6	±0	±0	±0	**+1**	±0	5	**+1**	**+1**	**+1**	**+1**	**+1**
satellite	9	±0	**+2**	±0	**+2**	±0	7	±0	±0	±0	**+1**	±0	6	±0	±0	±0	**+1**	±0	5	±0	±0	-1	**+1**	±0
scanalyzer08	13	**+1**	±0	±0	-1	-1	**12**	±0	±0	-3	±0	±0	**12**	±0	-3	-3	±0	±0	**12**	±0	-3	-3	±0	±0
scanalyzer11	**10**	±0	±0	±0	-1	-1	**9**	±0	±0	-3	±0	±0	**9**	±0	-3	-3	±0	±0	**9**	±0	-4	-3	±0	±0
sokoban08	**30**	±0	±0	±0	±0	±0	**29**	±0	±0	-1	±0	±0	24	±0	**+3**	±0	±0	±0	22	±0	**+3**	+1	±0	±0
sokoban11	**20**	±0	±0	±0	±0	±0	**20**	±0	±0	±0	±0	±0	**20**	±0	±0	±0	±0	±0	19	±0	**+1**	-1	±0	±0
storage	**20**	±0	±0	-1	-1	±0	**17**	±0	±0	-1	-1	±0	**15**	±0	±0	±0	±0	±0	**14**	±0	±0	±0	±0	±0
tetris14	**17**	±0	±0	±0	-15	-15	**14**	±0	-3	-4	-13	-12	**11**	-1	-3	-3	-11	-9	**9**	±0	-4	-4	-8	-7
tidybot11	**20**	±0	±0	±0	-19	-19	**20**	±0	-1	-3	-19	-19	**18**	-1	-4	-6	-17	-17	**13**	±0	-6	-8	-13	-12
tidybot14	**20**	±0	±0	±0	-20	-20	**18**	±0	-2	-5	-18	-18	**14**	-1	-6	-10	-14	-14	**6**	±0	-6	-6	-6	-6
tpp	**9**	±0	±0	±0	-1	±0	**7**	±0	±0	±0	-1	±0	**6**	±0	±0	±0	±0	±0	**6**	±0	±0	±0	±0	±0
transport08	17	±0	**+1**	-2	-1	±0	**15**	±0	±0	-1	-2	±0	12	**+1**	**+1**	-1	**+1**	**+1**	**11**	±0	±0	±0	-1	±0
transport11	15	±0	**+1**	-1	-2	-1	**11**	±0	±0	±0	-2	-1	8	**+1**	**+1**	-2	**+1**	**+1**	6	±0	±0	±0	**+1**	±0
transport14	13	**+1**	±0	-1	±0	±0	**9**	±0	±0	±0	-3	±0	**9**	±0	±0	-3	-2	±0	**7**	±0	-1	-2	-1	±0
trucks	**13**	-1	±0	-1	-1	-1	**8**	±0	±0	±0	±0	±0	**6**	±0	±0	±0	±0	±0	5	±0	±0	**+1**	±0	±0
visitall11	16	±0	**+1**	-1	±0	±0	**12**	-1	±0	-1	±0	±0	**9**	±0	±0	±0	±0	±0	**9**	±0	±0	±0	±0	±0
visitall14	**10**	±0	±0	-1	-1	±0	**6**	±0	±0	±0	±0	±0	**4**	±0	±0	±0	±0	±0	3	±0	±0	±0	**+1**	±0
woodwork08	**25**	±0	±0	-3	-6	-11	**15**	±0	-1	-3	-7	-4	10	±0	**+1**	-1	±0	-1	7	±0	**+2**	**+2**	**+2**	±0
woodwork11	**18**	±0	-1	-2	-3	-5	**10**	±0	-1	-3	-4	-4	5	±0	**+1**	-1	±0	-2	2	±0	+2	+2	**+3**	-1
zenotravel	**13**	±0	±0	±0	±0	±0	10	±0	±0	±0	**+2**	±0	8	±0	+1	±0	**+2**	±0	**8**	±0	±0	±0	±0	±0
Sum all	**1190**	-8	-20	-92	-139	-143	**897**	-5	-16	-85	-82	-84	**748**	-5	-22	-66	-22	-53	651	+7	-20	-39	**+13**	-26

Table 1: The coverage results as diff from the baseline BnB, for four domain suites defined by the 25%, 50%, 75%, and 100% of best known solution cost for the classical planning task as an OSP task cost bound. bl stands for blind, max^b and max for h^{max}, bound-sensitive and regular variants, ms^b and ms for *merge-and-shrink*, bound-sensitive and regular variants, respectively.

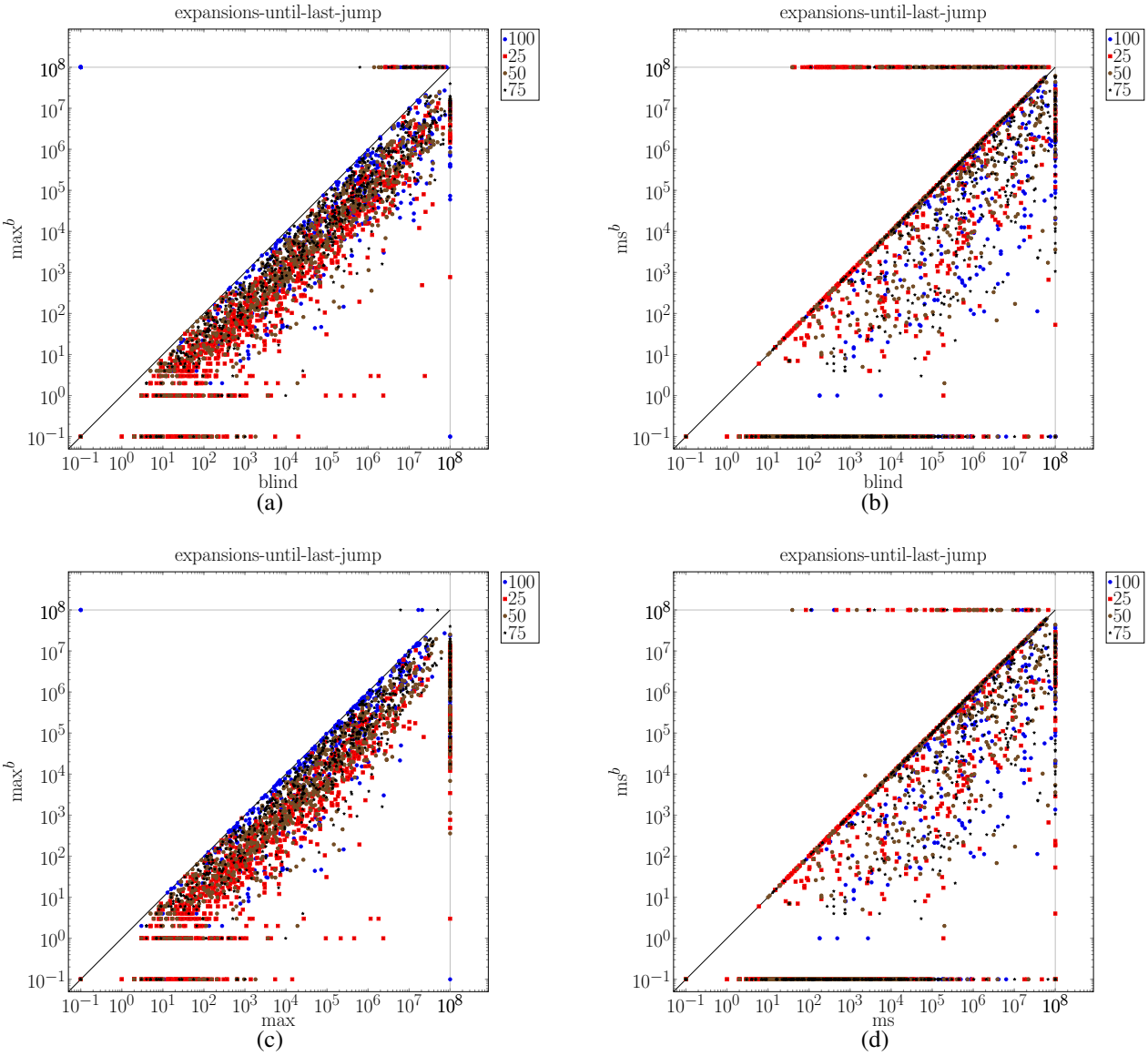


Figure 2: Expansions up to the last layer, A^* with blind heuristic vs. (a) bound-sensitive h^{max} and (b) bound-sensitive *merge-and-shrink*; A^* with bound-sensitive vs. regular heuristic for (c) h^{max} and (c) *merge-and-shrink*.

available OSP benchmarks [Katz *et al.*, 2019b]. The set of benchmarks is taken from the International Planning Competitions of recent years, in which goal facts are replaced with utilities, and the bound set at 25%, 50%, 75%, or 100% of the cost of the optimal or best known solution to each problem. The baseline for our comparison is a blind branch-and-bound search, currently the best available configuration for oversubscription planning that we know of [Katz *et al.*, 2019a]. We compare this baseline to our proposed approach of A^* search on the MCF compilation of the OSP task. Since the compilation introduces intermediate states at which some but not all of the $o^{v,\vartheta}$ have been applied, we use a further optimization that avoids generating these nodes and applies all of the $o^{v,\vartheta}$ actions in a single step, reducing the state space to that of the original OSP task. We experiment with blind A^* search, and A^* using classical h^{max} and h^{MS}, as well as the two heuristics' bound-sensitive variants introduced here. For h^{MS}, we used exact bisimulation with an abstract state space threshold of 50000 states and exact generalized label reduction [Sievers *et al.*, 2014]. The experiments were performed on Intel(R) Xeon(R) CPU E7-8837 @2.67GHz machines, with time and memory limits of 30min and 3.5GB, respectively. Per-domain and overall coverage, as well as per-task node expansions for the various configurations and problem suites are shown in Table 1 and Figure 2, respectively. We now report some observations from our results.

- Blind branch-and-bound search usually slightly outperforms blind A^* in terms of coverage, except for the 100% suite. The difference between the two may come

down to the fact that A^* must do extra work in ordering the priority queue, while the variant of branch and bound search that we consider uses no ordering heuristic and can use a simple stack as its search queue. Alternately it may be due to small differences in implementation.

- Bound-sensitive heuristics are much more informative than their classical variants on OSP problems, sometimes decreasing expansions by orders of magnitude. Compared to non-bound-sensitive heuristics, they also almost always result in better coverage.
- Blind search dominates informed search in terms of coverage when bounds are low, but the effect diminishes as the bound increases and it becomes intractable to explore the full state space under the bound. For the 25% suite of problems, heuristic configurations solve an average of approximately 100 instances fewer than the baseline, compared to approximately 15 instances fewer on the 100% suite. Notably, bound-sensitive h^{MS} has the best coverage in the 100% suite, solving 13 problems more than the baseline, and 6 more than blind A^*.
- Coverage on several domains benefits from more informed search schemes. On BLOCKSWORLD, DRIVERLOG, and MICONIC, bound-sensitive h^{MS} solves the largest number of problems, and this is also the case for bound-sensitive h^{max} on FLOORTILE, PARC-PRINTER, and SOKOBAN.
- h^{MS} often times out in the construction phase and before search has begun. This occurs on average in approximately 300 problems per suite, or 1200 problems total. This is especially pronounced in the TIDYBOT, TETRIS, and PIPESWORLD-NOTANKAGE domains. This suggests a hybrid approach that combines the strengths of blind search and h^{MS}: setting an upper bound on the time allotted to heuristic construction, and running blind search instead if construction does not terminate within this bound. Using this configuration with a value of 10 minutes for the upper bound results in a planner that outperforms blind A^* by $+11$, $+16$, $+37$, and $+38$ instances for the 25%, 50%, 75%, and 100% suites, respectively. This makes h^{MS} schemes that are less expensive to construct but maintain informativeness in this setting an appealing future subject of research.

Conclusions and Future Work

We have shown that a previously introduced compilation to multiple cost function classical planning allows the A^* algorithm to be used to solve oversubscription planning problems, and introduced a family of bound-sensitive heuristics that are much more informed than their classical counterparts in this setting. Our experiments show that this approach results in a state-of-the-art method for some bound settings and domains.

One future research direction we would like to explore that builds on the methods introduced here is the use of non-admissible heuristics for satisficing OSP. The method by which bound-sensitive h^{max} is obtained is fairly general and should be equally applicable for h^{add} or general relaxed plan heuristics [Keyder and Geffner, 2008]. A second direction is the use of these heuristics in other planning settings in which tradeoffs must be made between different cost functions, e.g. minimizing fuel use in the presence of bounds on time or vice versa in logistics problems.

Finally, our methods may be applicable to numeric planning problems in which the variables describe resources that are strictly decreasing and can be expressed in terms of secondary cost functions and associated bounds. Bound-sensitive heuristics could provide a principled way of reasoning about numeric variables in this context.

References

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

[Bonet and Geffner, 2001] Blai Bonet and Héctor Geffner. Planning as heuristic search. *AIJ*, 129(1):5–33, 2001.

[Borrajo *et al.*, 2013] Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini, editors. *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling (ICAPS 2013)*. AAAI Press, 2013.

[Dobson and Haslum, 2017] Sean Dobson and Patrik Haslum. Cost-length tradeoff heuristics for bounded-cost search. page 58, 2017.

[Edelkamp, 2001] Stefan Edelkamp. Planning with pattern databases. In Amedeo Cesta and Daniel Borrajo, editors, *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, pages 84–90. AAAI Press, 2001.

[Hart *et al.*, 1968] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[Haslum and Geffner, 2000] Patrik Haslum and Héctor Geffner. Admissible heuristics for optimal planning. In Steve Chien, Subbarao Kambhampati, and Craig A. Knoblock, editors, *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, pages 140–149. AAAI Press, 2000.

[Haslum, 2013] Patrik Haslum. Heuristics for bounded-cost search. In Borrajo et al. [2013], pages 312–316.

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

[Helmert *et al.*, 2014] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *JACM*, 61(3):16:1–63, 2014.

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

[Katz *et al.*, 2019a] Michael Katz, Emil Keyder, Florian Pommerening, and Dominik Winterer. Oversubscription planning as classical planning with multiple cost functions. In *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*. AAAI Press, 2019.

[Katz *et al.*, 2019b] Michael Katz, Emil Keyder, Florian Pommerening, and Dominik Winterer. PDDL benchmarks for oversubscription planning. `https://doi.org/10.5281/zenodo.2576024`, 2019.

[Keyder and Geffner, 2008] Emil Keyder and Héctor Geffner. Heuristics for planning with action costs revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, pages 588–592, 2008.

[Keyder and Geffner, 2009] Emil Keyder and Héctor Geffner. Soft goals can be compiled away. *JAIR*, 36:547–556, 2009.

[Mirkis and Domshlak, 2013] Vitaly Mirkis and Carmel Domshlak. Abstractions for oversubscription planning. In Borrajo et al. [2013], pages 153–161.

[Mirkis and Domshlak, 2014] Vitaly Mirkis and Carmel Domshlak. Landmarks in oversubscription planning. In Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan, editors, *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, pages 633–638. IOS Press, 2014.

[Muller and Karpas, 2018] Daniel Muller and Erez Karpas. Value driven landmarks for oversubscription planning. In Mathijs de Weerdt, Sven Koenig, Gabriele Röger, and Matthijs Spaan, editors, *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018)*, pages 171–179. AAAI Press, 2018.

[Sievers *et al.*, 2014] Silvan Sievers, Martin Wehrle, and Malte Helmert. Generalized label reduction for merge-and-shrink heuristics. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence (AAAI 2014)*, pages 2358–2366. AAAI Press, 2014.

[Smith, 2004] David E. Smith. Choosing objectives in over-subscription planning. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 393–401. AAAI Press, 2004.

[Stern *et al.*, 2011] Roni Tzvi Stern, Rami Puzis, and Ariel Felner. Potential search: A bounded-cost search algorithm. In Fahiem Bacchus, Carmel Domshlak, Stefan Edelkamp, and Malte Helmert, editors, *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling (ICAPS 2011)*, pages 234–241. AAAI Press, 2011.

[Thayer and Ruml, 2011] Jordan T. Thayer and Wheeler Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In Toby Walsh, editor, *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 674–679. AAAI Press, 2011.

[Trevizan *et al.*, 2017] Felipe W. Trevizan, Sylvie Thiébaux, and Patrik Haslum. Occupation measure heuristics for probabilistic planning. In Laura Barbulescu, Jeremy Frank, Mausam, and Stephen F. Smith, editors, *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)*, pages 306–315. AAAI Press, 2017.

Learning How to Ground a Plan – Partial Grounding in Classical Planning

Daniel Gnad, Álvaro Torralba
Saarland University
Saarland Informatics Campus
Saarbrücken, Germany
{gnad,torralba}@cs.uni-saarland.de

Martín Domínguez, Carlos Areces, Facundo Bustos
Universidad Nacional de Córdoba
Córdoba, Argentina
{mardom75, carlos.areces, facundojosebustos}@gmail.com

Abstract

Current classical planners are very successful in finding (non-optimal) plans, even for large planning instances. To do so, most planners rely on a preprocessing stage that computes a grounded representation of the task. Whenever the grounded task is too big to be generated (i.e., whenever this preprocess fails) the instance cannot even be tackled by the actual planner. To address this issue, we introduce a partial grounding approach that grounds only a projection of the task, when complete grounding is not feasible. We propose a guiding mechanism that, for a given domain, identifies the parts of a task that are relevant to find a plan by using off-the-shelf machine learning methods. Our empirical evaluation attests that the approach is capable of solving planning instances that are too big to be fully grounded.

Introduction

Given a model of the environment, classical planning attempts to find a sequence of actions that lead from an initial state to a state that satisfies a set of goals. Planning models are typically described in the Planning Domain Definition Language (PDDL) (McDermott et al. 1998) in terms of predicates and action schemas with arguments that can be instantiated with a set of objects. However, most planners work on a grounded representation without free variables, like STRIPS (Fikes and Nilsson 1971) or FDR (Bäckström and Nebel 1995). Grounding is the process of translating a task in the lifted (PDDL) representation to a grounded representation. It requires to compute all valid instantiations that assign objects to the arguments of predicates and action parameters, even though only a small fraction of these instantiations might be necessary to solve the task.

The size of the grounded task is exponential in the number of arguments in predicates and action schemas. Although this is constant for all tasks of a given domain, and grounding can be done in polynomial time, it may still be prohibitive when the number of objects is large and/or some predicates or actions have many parameters.

The success of planners like FF (Hoffmann and Nebel 2001a) or LAMA (Richter, Westphal, and Helmert 2011) in finding plans for large planning tasks is undeniable. However, since most planners rely on grounding for solving a task, they fail without even starting the search for a plan whenever an instance cannot be grounded, making grounding a bottleneck for the success of satisficing planners.

Grounding is particularly challenging in open multi-task environments, where the planning task is automatically generated with all available objects even if only a few of them are relevant to achieve the goals. For example, in robotics, the planning task may contain all objects with which the robot may interact even if they are not needed (Lang and Toussaint 2009). In network-security environments, like the one modeled in the Caldera domain (Miller et al. 2018), the planning task may contain all details about the network. However, to the best of our knowledge, no method exists that attempts to focus the grounding on relevant parts of the task.

We propose *partial grounding*, where, instead of instantiating the full planning task, we focus on the parts that are required to find a plan. The approach is sound – if a plan is found for the partially grounded task then it is a valid plan for the original task – but incomplete – the partially grounded task will only be solvable if the operators in at least one plan have been grounded. To do so, we give priority to operators that we deem more relevant to achieve the goal. Inspired by relational learning approaches to domain control knowledge (e.g., Yoon, Fern, and Givan (2008), de la Rosa et al. (2011), Krajnansky et al. (2014)), we use machine learning methods to predict the probability that a given operator belongs to a plan. We learn from small training instances, and generalize to larger ones by using relational features in standard classification and regression algorithms (e.g., Kramer, Lavrač, and Flach (2001)). As an alternative model, we also experiment with relational trees to learn the probabilities (Muggleton and Raedt 1994).

Empirical results show that our learning models can predict which operators are relevant with high accuracy in several domains, leading to a very strong reduction of task size when grounding and solving huge tasks.

Background

Throughout the paper, we assume for simplicity that tasks are specified in the STRIPS subset of PDDL (Fikes and Nilsson 1971). Our algorithms and implementation, however, are directly applicable to a larger subset of PDDL containing ADL expressions (Pednault 1989).

A lifted (PDDL) task Π^{PDDL} is a tuple $(\mathcal{P}, \mathcal{A}, \Sigma^C, \Sigma^O, I, G)$ where $\mathcal{P}$ is a set of (first-order) atomic

predicates, $\mathcal{A}$ is a set of *action schemas*, $\Sigma := \Sigma^C \cup \Sigma^O$ is a non-empty set of *objects* consisting of constants Σ^C, and non-constant objects Σ^O, I is the *initial state*, and G is the *goal*. Predicates and action schemas have parameters. We denote individual parameters with x, y, z and sets of parameters with X, Y, Z. An action schema $a[X]$ is a triple $(\mathsf{pre}(a), \mathsf{add}(a), \mathsf{del}(a))$, consisting of *preconditions*, *add list*, and *delete list*, all of which are subsets of $\mathcal{P}$, possibly pre-instantiated with objects from Σ^C, such that X is the set of variables that appear in $\mathsf{pre}(a) \cup \mathsf{add}(a) \cup \mathsf{del}(a)$. I and G are subsets of $\mathcal{P}$, instantiated with objects from Σ.

A lifted task Π^{PDDL} can be divided into two parts: the domain specification $(\mathcal{P}, \mathcal{A}, \Sigma^C)$ which is common to all instances of the domain, and the problem specification (Σ^O, I, G) which is different for each instance of a domain.

A STRIPS task Π is a tuple (F, O, I, G), where F is a set of grounded predicates, called *facts*, and O is a set of grounded action schemas, called *operators*. A *state* $s \subseteq F$ is a set of facts, $I \subseteq F$ is the *initial state* and $G \subseteq F$ is the *goal*. An operator o is *applicable* in a state s if $\mathsf{pre}(o) \subseteq s$. In that case, the outcome state is $s' = (s \setminus \mathsf{del}(o)) \cup \mathsf{add}(o)$, and we write $s \xrightarrow{o} s'$ for the transition from s to s' via o. For a sequence of operators $\overline{o}$, we write $s \xrightarrow{\overline{o}} t$ if the operators in $\overline{o}$ can be iteratively applied to s, resulting in t. A sequence $\overline{o}$, with $I \xrightarrow{\overline{o}} s_G$, is a *plan* for Π if $G \subseteq s_G$. A task Π is solvable if a plan exists. The plan is *optimal* if its length is minimal among all plans for Π.

We define the *delete-relaxation* of a task Π as the task Π^+ obtained by setting $\mathsf{del}(o) = \emptyset$, for all $o \in O$. We say that Π is *delete-relaxed solvable* if Π^+ is solvable.

Given a lifted task Π^{PDDL}, we can compute the corresponding STRIPS task Π by *instantiating* the predicates and action schemas with the objects in Σ. Then, F contains a fact for each possible assignment of objects in Σ to the arguments of each predicate $P[X] \in \mathcal{P}$, and O contains an operator for each possible assignment of objects in Σ to each action schema $a[X] \in \mathcal{A}$. In practice, we do not enumerate all possible assignments of objects in Σ to the arguments in facts and action schemas. Instead, only those facts and operators are instantiated that are delete-relaxed reachable from the initial state (Helmert 2009).

Partial Grounding

We base our method on the grounding algorithm of Fast Downward (Helmert 2006). To ground a planning task, this algorithm performs a fix-point computation similar to the computation of relaxed planning graphs (Blum and Furst 1997), where a queue is initialized with the facts in the initial state and in each iteration one element of the queue is popped and processed. If the element is a fact, then those operators of which all preconditions have already been processed (are reached) are added to the queue. If the element is an operator, all its add effects are pushed to the queue. The algorithm terminates when the queue is empty. Then, all processed facts and operators are delete-relaxed reachable from the initial state. For simplicity, the algorithm we describe here considers only STRIPS tasks but it can be adapted to support other PDDL features like negative preconditions or conditional effects as it is done by Helmert (2009).

Algorithm 1: Partial Grounding.

Input: A lifted task $\Pi^{PDDL} = (\mathcal{P}, \mathcal{A}, \Sigma^C, \Sigma^O, I, G)$
Output: A STRIPS task $\Pi = (F, O, I, G)$

```
1  q ← I;
2  F ← ∅ ;                    // Processed facts
3  O ← ∅ ;                    // Processed operators
4  while ¬(q.empty() ∨ G ⊆ F) ∧ ¬StoppingCondition
     do
5      if q.containsFact() then
6          f ← q.popFact() ;
7          F ← F ∪ {f} ;
8          for o ∉ O ∧ pre(o) ⊆ F do
9              q.insert(o) ;
10     else
11         o ← q.popHighPriorityOperator() ;
12         O ← O ∪ {o} ;
13         for f ∉ F ∧ f ∈ add(o) do
14             q.insert(f) ;
15 return (F, O, I, G)
```

Algorithm 1 shows details of our approach. The main difference with respect to the approach by Helmert (2009) is that (1) the algorithm can stop before the queue is empty, and (2) operators are instantiated in a particular order. For these two choice points we suggest an approach that aims at minimizing the size of the partially grounded task, while keeping it solvable. That said, our main focus is the operator ordering, and we only consider a simple stopping condition.

Stopping condition. Typical grounding approaches terminate only when the queue is empty, meaning that all (delete-relaxed) reachable facts and operators have been grounded. In partial grounding, we allow the algorithm to stop earlier. Intuitively, this is a good idea because most planning tasks have short plans, usually in the order of at most a few hundred operators, compared to possibly millions of grounded operators. Hence, if the correct operators are selected, partial grounding can potentially stop much sooner than complete grounding. The key issue is how to decide when the probability of finding a plan using the so-far grounded operators is sufficient. Consider the following claims:

1. The grounded task is delete-relaxed solvable iff $G \subseteq F$.
2. The grounded task is solvable iff there exists a plan π for Π^{PDDL} such that $\pi \subseteq O$.

Item 1 provides a necessary condition for the task to be relaxed-solvable, so grounding should continue at least until $G \subseteq F$. But this is not sufficient, as it does not guarantee that a plan can be found for the non-relaxed task. Item 2 provides an obvious, but difficult to predict, condition for success.

In this work, we consider only a simple stopping condition. To maximize the probability of the task being solvable, it is desirable to ground as many operators as possible. The main constraint on the number of operators to ground are the

resources (time and memory) that can be spent on grounding. For that reason, one may want to continue grounding while these resources are not compromised[1]. We provide a constant N^{op} as a parameter, an estimate on the number of operators that can be grounded given the available resources, and let the algorithm continue as long as $|O| \leq N^{op}$.

If not all actions are grounded, the resulting grounded task is a partial representation of the PDDL input and the overall planning process of grounding and finding a plan for the grounded task is incomplete. We implemented a loop around the overall process that incrementally grounds more actions, when finding the partially grounded task unsolvable. This converges to full grounding, resulting in a complete planner.

Queue order. Standard grounding algorithms extract elements from the queue in an arbitrary order – since all operators are grounded, order does not matter. Our algorithm always grounds all facts that have been added to the queue, giving them preference over operators. This ensures that the effects of all grounded operators are part of the grounded task. After all facts in the queue have been processed, our algorithm picks an operator according to a heuristic criterion, which we will call the *priority function*. Some simple priority functions include FIFO, LIFO, or random. Since our aim is to ground all operators of a plan, the priority queue should sort operators by their probability of belonging to a plan. To estimate these probabilities, we use machine learning techniques as detailed in the next section. Additionally, one may want to increase the diversity of selected operators to avoid being misguided by a bias in the estimated probabilities. We consider a simple *round robin* (RR) criterion, which classifies all operators in the queue by the action schema they belong to, and chooses an operator from a different action schema in each iteration. RR works in combination with a priority function that is used to select which instantiation of a given action schema should be grounded next.

We define a *novelty* criterion as a non-trivial priority function that is not based on learning, inspired by novelty pruning techniques that have successfully been applied in classical planning (Lipovetzky and Geffner 2012; 2017). During search, the novelty of a state is defined as the minimum number m for which the state contains a set of facts of size m, that is not part of any previously generated state. This can be used to prune states with a novelty $< k$.

We adapt the definition of novelty to operators in the grounding process as follows. Let Σ be the set of objects, $a[X]$ an action schema, and O the set of already grounded operators corresponding to all instantiations of $a[X]$. Let $\sigma = \{(x_1, \sigma_1), \ldots, (x_k, \sigma_k)\}$ be an assignment of objects in Σ to parameters X instantiating an operator o, such that $o \notin O$. Then, the novelty of o is defined as the number of assignments (x_i, σ_i) such that there does not exist an operator $o' \in O$ where x_i got assigned σ_i. In the grounding we will prioritize operators with a higher novelty, which are likely to generate facts that have not been grounded yet.

[1]While search can benefit from grounding less operators, an orthogonal pruning method that uses full information of the grounded task, can be employed at that stage (e. g. Heusner et al. (2014)).

Learning Operator Priority Functions

To guide the grounding process towards operators that are relevant to solve the task, we use a priority queue that gives preference to more promising operators. We use a priority function $f : O \to [0, 1]$ that estimates whether operators are useful or not. Ideally, we want to assign 1 to operators in an optimal plan and 0 to the rest, so that the number of grounded operators is minimal. We approximate this by assigning to each operator a number between 0 and 1 that estimates the probability that the operator belongs to an optimal plan for the task. This is challenging, however, due to lack of knowledge about the fully grounded task.

We use a learning approach, training a model on small instances of a domain and using it to guide grounding in larger instances. Our training instances need to be small enough to compute the set of operators that belong to any optimal plan for the task. We do this by solving the tasks with a symbolic bidirectional breadth-first search (Torralba et al. 2017) and extracting all operators that belong to an optimal solution.

Before grounding, the only information that we have available is the lifted task $\Pi^{PDDL} = (\mathcal{P}, \mathcal{A}, \Sigma^C, \Sigma^O, I, G)$. Our training data uses this information, consisting of tuples $(I, G, \Sigma^O, o, \{0, 1\})$ for each operator o in a training instance, where o is assigned a value of 1 if it belongs to an optimal solution and 0 otherwise. We formulate our priority functions as a classification task, where we want to order the operators according to our confidence that they belong to the 1 class. To learn a model from this data, we need to characterize the tuple (I, G, Σ^O, o) with a set of features. Since training and testing problems have different objects, these features cannot refer to specific objects in Σ^O, so learning has to be done at the lifted level. We propose relational rules that connect the objects that have instantiated the action schema to the training sample (I, G, Σ^O) to capture meaningful properties of an operator. Because different action schemas have different (numbers of) arguments, the features that characterize them will necessarily be different. Therefore, we train a separate model for each action schema $a[X] \in \mathcal{A}$. All these models, however, predict the probability of an operator being in an optimal plan, so the values from two different models are still comparable.

We considered two approaches to conduct the learning: inductive relational trees and classification/regression with relational features.

Inductive Relational Learning Trees. Inductive Logic Programming (ILP) (Muggleton and Raedt 1994) is a well-known machine learning approach suitable when the training instances are described in relational logic. ILP has been used, e.g., to learn domain control knowledge for planning (de la Rosa et al. 2011; Krajnansky et al. 2014). We use the Aleph tool (Srinivasan 1999) to learn a tree where each inner node is a predicate connecting the parameters of a to-be-grounded operator to the facts in the initial state or goal, to objects referred to in a higher node in the tree, or to a constant. The nodes are evaluated by checking if there exists a predicate instantiated with the given objects in the initial state or goal. A wildcard symbol ("_") indicates that we do

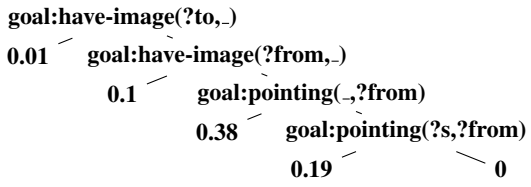


Figure 1: The relational tree learned for the action schema *turn-to(?s - satellite, ?from - direction, ?to - direction)* in the Satellite domain.

not require a particular object, but that any object instantiating the predicate at this position is fine. In Figure 1, the left child corresponds to this check evaluating to false and the right child to true. For a given action, the tree is evaluated by checking if there exists an assignment to the free variables in a path from the root to a leaf node, such that all nodes on the path evaluate to the correct truth value. We then take the real value in the leaf node as an estimate of the probability that the operator is part of an optimal plan. This evaluation is akin to a CSP problem, so we need to keep the depth of the trees at bay to have an efficient priority function.

Figure 1 shows the tree learned for the *turn-to* action schema in Satellite. In this domain, the goal is to take pictures in different modes. Several satellites are available, each with several instruments that support some of the modes. The actions are switching the instruments on and off, calibrating them, turning the satellite into different directions, and taking images. The *turn-to* action changes the direction satellite $?s$ is looking at. In this case, the learned tree considers that the operators turning to and from relevant directions are more likely part of an optimal plan than turning away from the goal direction. More concretely, if for a to-be-instantiated operator *turn-to(s, from, to)* with objects *s*, *from*, and *to*, there is no goal *have-image(to, _)*, i. e., taking an image in direction *to* using any mode, then the operator is deemed not useful by the trained model, it only has a probability of 1% of belonging to an optimal plan. In the opposite case, and if there is a *have-image* goal in the *from* direction, but no goal *pointing(_, from)*, then the operator is expected to be most useful, with a probability of 38% of being in an optimal plan. This is relevant information to predict the usefulness of *turn-to*. However, there is some margin of improvement since the initial state is ignored.

Classification and Regression with Relational Features. An alternative is to use relational rules as features for standard classification and regression algorithms (Kramer, Lavrač, and Flach 2001). Our features are relational rules where the head is an action schema, and the body consists of a set of goal or initial-state predicates, partially instantiated with the arguments of the action schema in the head of the rule, constant objects in Σ^C, or free variables used in other predicates in the rule. This is very similar to a path from root to leaf in the aforementioned relational trees.

We generate rules by considering all possible predicates and parameter instantiations with two restrictions. First, to guarantee that the rule takes different values for different instantiations of the action schema, one of the arguments in the first predicate in the body of the rule must be bound to a parameter of the action schema. Second, at least one argument of each predicate after the first one, must be bound to a free variable used in a previously used predicate. This aims at reducing the number of features by avoiding redundant rules that can be described as a conjunction of simpler rules. We assume that, if the conjunction of two rules is relevant for the classification task, the machine learning algorithms will be able to infer this.

Most of the generated rules do not provide useful information to predict whether an operator will be part of an optimal plan or not. This is because we brute-force generate all possible rules, including many that do not capture any useful properties. Therefore, it is important to select a subset of relevant features. We do this filtering in two steps. First, we remove all rules that evaluate to the same value in all training instances (e.g., rules that contain `goal:predicate` in the body will never evaluate to true if `predicate` is never part of the goal description in that domain). Then, we use attribute selection techniques in order to filter out those features that are not helpful to predict whether the operator is part of an optimal plan. As an example, the most relevant rule generated for the *turn-to* schema is:

```
turn-to(?s, ?to, ?from) :- goal:have-image(?to, ?M1),
    goal:have-image(?from,?M2), ini:on-board(?I, ?s),
    ini:supports(?I, ?M1), ini:supports(?I, ?M2).
```

This can be read as: "do we have to take images in directions *?to* and *?from* in modes that are supported by one of the instruments on board?". This rule surprisingly accurately describes a scenario where *turn-to* is relevant (and can be complemented with other rules to capture different cases).

Given a planning task and an operator, a rule is evaluated by replacing the arguments in the head of the rule by the objects that are used to instantiate the operator and checking if there exists an assignment to the free variables such that the corresponding facts are present in the initial state and goal of the task. Doing so, we generate a feature vector for each grounded action from the training instances with a binary feature for every rule indicating whether the rule evaluates to true for that operator or not. This results in a training set where for each operator we get a vector of boolean features (one feature per rule), together with a to-be-predicted class that is 1 if the operator is part of an optimal plan for the task, and 0 if not. On this training set, we can use either classification or regression methods to map each operator to a real number. With classification methods we use the confidence that the model has in the operator belonging to the positive class. In regression, the model directly tries to minimize the error by assigning values to 1 for operators in an optimal plan and 0 to others. It is important to note that it is possible that there are two training examples with the same feature vector, but with different values in the target. In these cases, we merge all training examples with the same feature vector and replace them with a single one that belongs to the 1 class if any of the examples did[2].

[2]For regression algorithms, we also considered taking the aver-

During grounding, for every operator that is inserted in the queue, we evaluate all rules and call the model to get its priority estimate. To speed-up rule evaluation, we precompute, before grounding, all possible assignments to the arguments of the action schema that satisfy the rule. The computational cost of doing this is exponential in the number of free variables but it was typically negligible for the rules used by our models. We evaluate the relational trees in a similar way.

Experiments

For the evaluation of our partial grounding approach, we adapted the implementation of the "translator" component of the Fast Downward planning system (FD) (Helmert 2006). The translator parses the given input PDDL files and outputs a fully grounded task in finite-domain representation (FDR) (Bäckström and Nebel 1995; Helmert 2009) that corresponds to the PDDL input. Our changes are minimally invasive, only changing the ordering in which actions are handled and the termination condition, as indicated in Algorithm 1. Therefore, none of the changes affect the correctness of the translator, i. e., the generated grounded planning task will always be a proper FDR task. The changes do not affect the performance too much either, except when using a computationally expensive priority function.

Experimental Setup. For the evaluation of our technique, we require domains for which (1) instance generators are available to generate a set of diverse instances small enough for training, and (2) the size of the grounded instances grows at least cubically with respect to the parameters of the generator so that we have large instances that are hard to fully ground, for evaluation. We picked four domains that were part of the learning track of the international planning competition (IPC) 2011 (Blocksworld, Depots, Satellite, and TPP), as well as two domains of the deterministic track of IPC'18 (Agricola and Caldera). For all domains, we used the deterministic track IPC instances and a set of 25 large instances that we generated ourselves for the experiments.

For the training of the models, we used between 40 and 250 small instances, to get enough training data for each action schema. Since the number of grounded actions per schema varies significantly across domains, we individually adapted the number of training instances.

To generate the large instances, we started at roughly the same size as the largest IPC instances, scaling the parameters of the generator linearly when going beyond that. As an example, in Satellite, the biggest IPC instance has around 10 satellites and 20 instruments, which is the size of our smallest instances. In the largest instances that we generated, there are up to 15 satellites and 60 instruments. In Blocksworld, where IPC instances only scale up to 17 blocks, we scale in a different way, starting at 75 blocks and going up to 100, which can still easily be solved by our techniques.

Regarding the domains, we used the typed domain encoding of Satellite from the learning track, which simplifies rule generation, but does not semantically change the domain. In Blocksworld, we use the "no-arm" encoding, which shows a cubic explosion in the grounding, in contrast to the "arm" encoding, where the size of the grounded task is only quadratic in the PDDL description.

age but this resulted in slightly worse results in most cases.

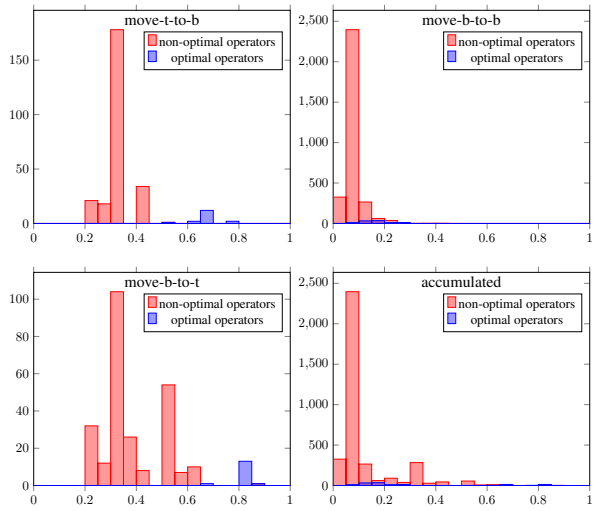


Figure 2: Evaluation of logistic regression in Blocksworld.

Beside the static queue orderings, FIFO, LIFO, and novelty-based methods, we experiment with learning-based approaches using classification and regression models. While the former exemplify what is possible without learning, the latter methods aim at grounding only those actions that belong to a plan for a given task. In all cases, we combine the methods with the round robin queue setup (**RR**).

Learning Framework. We report results for a *logistic regression classifier* (**LOGR**), *kernel ridge regression* (**KRN**), *linear regression* (**LINR**), and a *support vector machine regressor* (**SVR**). While LINR and LOGR learn linear functions to combine the features, differing mostly in the loss function and underlying model that is used, KRN and SVR are capable of learning non-linear functions. We expect non-linear functions to be useful to combine features, which cannot be done with linear functions. We also report the results of the decision trees learned by **Aleph**. To implement the machine learning algorithms, we use the `scikit` Python package (Pedregosa et al. 2011), which nicely connects to the translator in FD.

For feature selection, i. e., to select which rules are useful to predict the probability of an operator to be in a plan, we used the properties included in the trained models. For each feature (rule) contained in the feature vector, the model returns a weight according to its relevance to discriminate the target vector. After experimenting with multiple different models, we decided to use a decision tree regressor to predict rule usefulness, for all trained models.

We evaluated our models in isolation on a set of validation instances that are distinct from both our training and testing set, and small enough to compute the set of operators that are part of any optimal plan. Figure 2 shows the outcome of the priority function learned by LOGR in Blocksworld. The

			Solved within 30min overall															Last iteration solved within 30min													
						Novelty		LINR		LOGR		KRN		SVR		Aleph					Novelty		LINR		LOGR		KRN		SVR		Aleph	
Domain	#	Base	FIFO	LIFO	RND		RR		RR		RR		RR		RR		RR	FIFO	LIFO	RND		RR		RR		RR		RR		RR		RR
Agricola-IPC	20	**10**	1	1	2	1	3	1	7	5	5	8	5	4	**10**	**10**	3	9	9	9	7	10	9	**12**	11	**12**	9	**12**	10	**12**	11	10
Agricola-large	25	4	0	0	0	0	0	0	1	2	1	0	1	0	**22**	17	0	4	4	4	0	10	1	**24**	20	23	6	**24**	19	**24**	**24**	**24**
Blocksworld-IPC	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35
Blocksworld-large	25	0	0	0	0	21	**25**	14	**25**	**25**	23	**25**	24	**25**	22	**25**	**25**	0	0	0	21	**25**	14	**25**	**25**	24	**25**	**25**	**25**	24	**25**	**25**
Caldera-IPC	20	13	13	12	13	9	14	17	18	18	18	11	18	**19**	18	13	14	17	13	17	15	17	19	19	19	19	17	19	**20**	19	15	18
Caldera-large	25	0	10	0	3	0	5	22	18	18	**23**	1	19	20	17	0	7	19	0	5	12	16	**25**	**25**	24	**25**	8	**25**	**25**	**25**	0	19
Depots-IPC	22	20	19	20	19	19	20	20	20	19	**21**	19	20	20	**21**	20	20	19	20	19	19	20	20	20	19	**21**	19	20	20	**21**	20	20
Depots-large	25	1	0	0	0	0	0	**5**	3	1	2	2	3	1	4	2	0	0	0	0	0	0	**5**	3	1	2	2	3	1	4	2	0
Satellite-IPC	36	**36**	35	**36**	**36**	35	26	**36**	35	**36**	35	35	35	**36**	35	**36**	**36**	35	**36**	**36**	**36**	35	**36**	**36**	**36**	**36**	**36**	**36**	**36**	**36**	**36**	**36**
Satellite-large	25	0	0	0	0	1	0	0	11	0	14	**15**	14	0	14	1	1	0	0	0	1	0	0	14	0	16	**19**	15	0	16	1	1
TPP-IPC	30	**30**	**30**	**30**	**30**	**30**	**30**	26	28	**30**	**30**	**30**	**30**	**30**	29	**30**	**30**	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
TPP-large	25	7	5	**8**	2	6	**8**	1	2	4	4	5	6	4	6	**8**	6	6	8	6	6	8	5	5	7	5	5	6	7	**9**	8	6
$\sum$	313	156	148	142	140	157	166	177	203	193	211	186	210	194	**233**	197	177	174	155	161	182	206	199	248	227	248	211	250	228	**255**	207	224

Table 1: Number of instances solved by the baseline with full grounding (Base), and incremental grounding with static action orderings (FIFO, LIFO, random (RND)), novelty-based ordering, and several learning models (see text). "RR" indicates that we use a separate priority queue for each action schema, taking turns over the schemas. Best coverage highlighted in **bold** face.

bars indicate the number of operators across all validation instances that got a priority in a given interval, highlighting operators from optimal plans in a different color. The plots nicely illustrate that the priority function works very well for the action schemas *move-t-to-b* and *move-b-to-t*, where it is able to distinguish "optimal" from "non-optimal" operators. The distinction works not so well for *move-b-to-b*, but in general gives a significantly lower priority to this action schema. Another important observation is that the total number of grounded *move-b-to-b* actions is much higher than that of the other two action schemas.

Projecting these observations to the grounding process, we expect the model to work well when used in a single priority queue, since it will prioritize *move-t-to-b* and *move-b-to-t* (which are the only ones needed to solve any Blocksworld instance) over *move-b-to-b* (which is only needed to optimally solve a task). On the validation set, grounding all operators with a priority of roughly > 0.6 suffices to solve the tasks, pruning all *move-b-to-b* operators and most non-optimal ones of the other schemas. RR in contrast will ground an equal number of all action schemas, including many unnecessary operators. These conjectures are well-supported by the plots in Figure 3.

When working with machine learning techniques, there is always the risk of overfitting. In our case the results on the training set are very similar to those on the validation set shown in Figure 2, suggesting that overfitting is not an issue in our setup. The results in other domains are similar.

Incremental Grounding. We use the incremental approach, where the first iteration grounds a given task until the goal is found to be relaxed-reachable. The left half of Figure 3 shows detailed information on how many operators need to be grounded until this is achieved for different priority functions. We discuss details later. In case this first iteration fails, i. e., the partial task is not solvable, we set a minimum number of operators to be grounded in the next iteration by using an increment of 10 000 operators. This strategy does not aim to maximize coverage but rather to find out what is the minimum number of operators that need to be grounded to solve a task for each priority function (with a granularity of 10 000 operators). The number of operators that were necessary to actually solve a given instance is illustrated in the right half of Figure 3.

For all configurations, after grounding, we run the first iteration of the LAMA planner (Richter and Westphal 2010), a good standard configuration for satisficing planning that is well integrated in FD. We also use LAMA's first iteration as a baseline on a fully grounded task, with runtime and memory limits for the entire process of 30 minutes and 4GB. All other methods perform incremental grounding using their respective priority function. We allowed for a total of 5 hours and 4GB for the incremental grounding, while restricting the search part to only 10 minutes per iteration to keep the overall runtime of the experiments manageable.

We show coverage, i. e., number of instances solved, in Table 1, with the time and memory limits mentioned in the previous subsection. The left part of the table considers instances as solved when the overall incremental grounding process (including finding a plan) finished within 30 min. In the right part, we approximate the results that could be achieved with a perfect stopping condition by considering an instance as solved if the last iteration, i. e., the successful grounding and search, finished within 30 min.

The baseline (**Base**) can still fully ground most instances except in Caldera and TPP, but fails to solve most of the large instances with up to 9 million operators. We scaled instances in this way so that a comparison of the number of grounded operators to the baseline is possible; further scaling would make full grounding impossible.

The table nicely shows that the incremental grounding approach, where several iterations of partial grounding and search are performed (remember that we only allow 10min for the search), significantly outperforms the baseline, even when considering an overall time limit of 30min. In fact, all instances in Blocksworld can be solved in less than 10s

by LOGR. This illustrates the power of our approach when the learned model captures the important features of a domain. The static orderings typically perform worse than the baseline, only the novelty-based ordering can solve more instances in Blocksworld, and in Caldera when using RR.

The plots in Figure 3 shed further light on the number of operators when (leftmost two columns) the goal is relaxed reachable in the first iteration and (rightmost two columns) the number of operators needed to actually solve the task. Each data point corresponds to a planning instance, with the number of ground actions of a fully grounded task on the x-axis. The y-axis shows the number of grounded actions for several priority functions, including FIFO (LIFO in TPP), novelty, the learned model that has the highest reduction on the number of grounded actions, and Aleph.

In general, the models capture the features of most domains quite accurately, leading to a substantial reduction in the size of the grounded task, and still being able to find a solution. The plots show that our models obtain a very strong reduction of the number of operators in the partially grounded task in Agricola, Blocksworld, and Caldera; some reduction (one order of magnitude) in Depots, and Satellite, and a small reduction in TPP. In terms of the size of the partially grounded tasks, different learning models perform best in different domains, and there is not a clear winner. In comparison, the baselines FIFO, LIFO, and Random do not significantly reduce the size of the grounded task in most cases, with a few exceptions like LIFO in TPP and FIFO in Caldera. The novelty criterion is often the best method among those without learning.

Grounding a delete-relaxed reachable task with less operators is often beneficial, but may be detrimental for the coverage if the task is unsolvable, as happens for the Novelty method in Agricola or the LIFO method in TPP. This also explains why the learning models with highest reductions in some domains (e.g. LOGR in Agricola) are not always the same as the ones with highest coverage. The RR queue mechanism often grounds more operators before reaching the delete-relaxed goal but this makes the first iteration solvable more often leading to more stable results. The exception is Aleph, where RR has the opposite effect, making the partially grounded tasks unsolvable.

Related Work

Some approaches in the literature try to alleviate the grounding problem, e. g. by avoiding grounding facts and operators unreachable from the initial state (Helmert 2009), reformulating the PDDL description by splitting action schemas with many parameters (Areces et al. 2014), or using symmetries to avoid redundant work during the grounding process (Röger, Sievers, and Katz 2018).

Lifted planning approaches that skip grounding entirely (Penberthy and Weld 1992) have lost popularity due to the advantages of grounding to speed-up the search and allow for more informative heuristics which are not easy to compute in a lifted level. Ridder and Fox (2014) adapted the delete-relaxation heuristic (Hoffmann and Nebel 2001a) to the lifted level. This is related to our partial grounding approach since their relaxed plan extraction mechanism can be used to obtain a grounded task where the goal is relaxed reachable, and it could be used to enhance the novelty and learning priority functions that we use here.

There are many approaches to eliminate irrelevant facts and operators from grounded tasks (Nebel, Dimopoulos, and Koehler 1997; Hoffmann and Nebel 2001b; Haslum, Helmert, and Jonsson 2013; Torralba and Kissmann 2015). The closest to our approach is under-approximation refinement (Heusner et al. 2014), which also performs search with a subset of operators. However, all these techniques use information from the fully grounded representation to decide on the subset of relevant operators, so are not directly applicable in our setting. The results of our learning models (see Figure 2) show that applying learning to identify irrelevant operators is a promising avenue for future research.

Recently, Toyer et al. (2018) introduced a machine learning approach to learn heuristic functions for specific domains. This is similar to our work, in the sense that a heuristic estimate has been learned, though for states in the search, not actions in the grounding. Furthermore, the authors used neural networks instead of our, more classical, models.

Conclusion

In this paper, we proposed an approach to partial grounding of planning tasks, to deal with tasks that cannot be fully grounded under the available time and memory resources. Our algorithm heuristically guides the grounding process giving preference to operators that are deemed most relevant for solving the task. To determine which operators are relevant, we train different machine learning models using optimal plans from small instances of the same domain. We consider two approaches, a direct application of relational decision trees, and using relational features with standard classification and regression algorithms. The empirical results show the effectiveness of the approach. In most domains, the learned models are able to identify which operators are relevant with high accuracy, helping to reduce the number of grounded operators by several orders of magnitude, and greatly increasing coverage in large instances.

Acknowledgments This work was supported by the bilateral project of the German Academic Exchange Service (DAAD) and the Argentinian Ministry of Science, Technology, and Productive Innovation (MinCyT) number DA/16/01 "Optimizing Planning Domains". Daniel Gnad was partially supported by the German Research Foundation (DFG), under grant Nr. HO 2169/6-1, "Star-Topology Decoupled State Space Search".

References

Areces, C.; Bustos, F.; Dominguez, M.; and Hoffmann, J. 2014. Optimizing planning domains by automatic action schema splitting. In *Proc. ICAPS'14*.

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

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1–2):279–298.

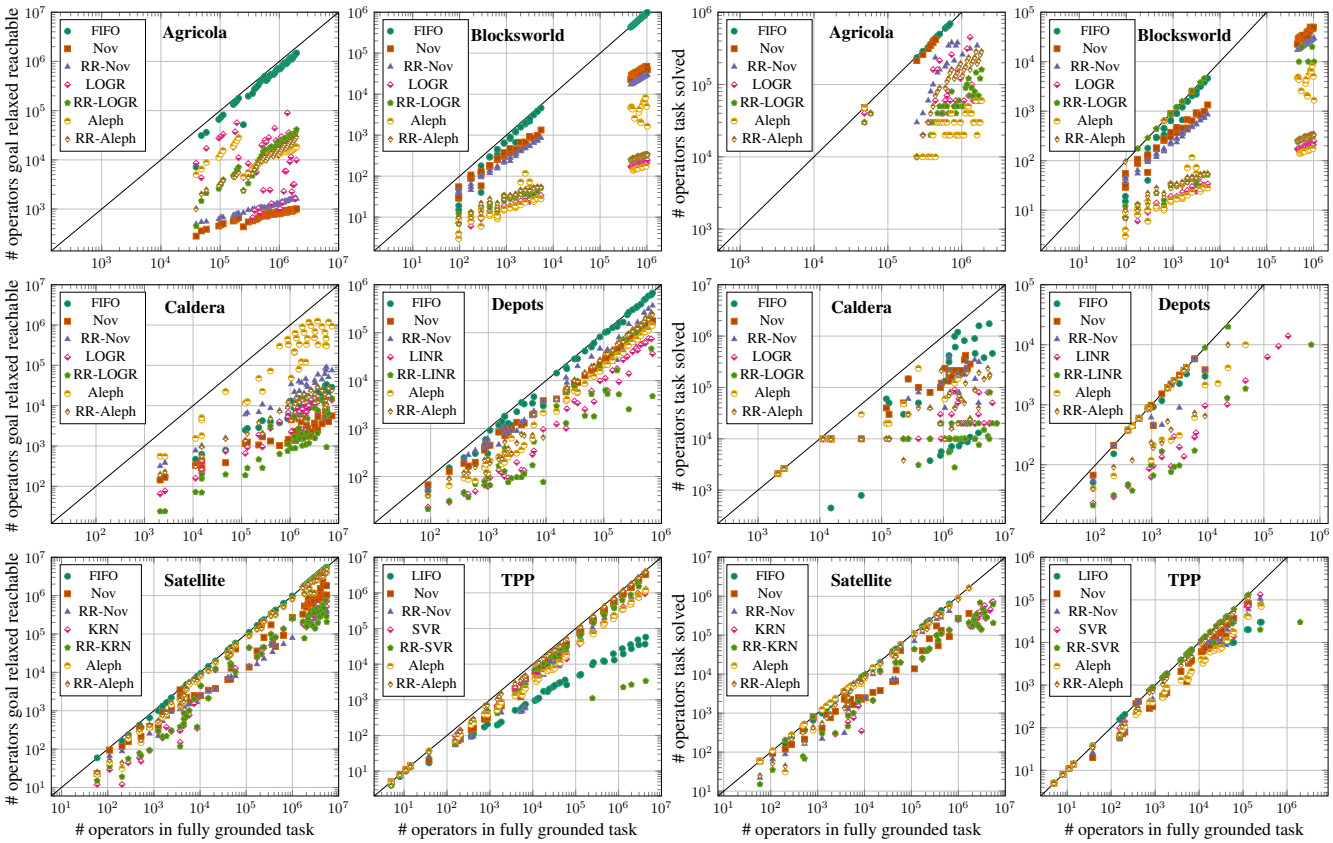


Figure 3: The scatter plots show the number of operators of a fully grounded task on the x-axis. The y-axis shows the number of operators that are needed to make the goal reachable in the grounding (leftmost two columns), and the number of operators that are needed to solve the task (rightmost two columns), for several priority functions.

de la Rosa, T.; Celorrio, S. J.; Fuentetaja, R.; and Borrajo, D. 2011. Scaling up heuristic planning with relational decision trees. *JAIR* 40:767–813.

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

Haslum, P.; Helmert, M.; and Jonsson, A. 2013. Safe, strong, and tractable relevance analysis for planning. In *Proc. ICAPS'13*.

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

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

Heusner, M.; Wehrle, M.; Pommerening, F.; and Helmert, M. 2014. Under-approximation refinement for classical planning. In *Proc. ICAPS'14*.

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

Hoffmann, J., and Nebel, B. 2001b. RIFO revisited: Detecting relaxed irrelevance. In *Proc. of ECP'01*, 325–336.

Krajnansky, M.; Buffet, O.; Hoffmann, J.; and Fern, A. 2014. Learning pruning rules for heuristic search planning. In *Proc. of ECAI'14*, 483–488.

Kramer, S.; Lavrač, N.; and Flach, P. 2001. Propositionalization approaches to relational data mining. In *Relational data mining*. Springer. 262–291.

Lang, T., and Toussaint, M. 2009. Relevance grounding for planning in relational domains. In *Proc. of ECML'09*, 736–751.

Lipovetzky, N., and Geffner, H. 2012. Width and serialization of classical planning problems. In *Proc. of ECAI'12*, 540–545.

Lipovetzky, N., and Geffner, H. 2017. A polynomial planning algorithm that beats LAMA and FF. In *Proc. ICAPS'17*, 195–199.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Comitee.

Miller, D.; Alford, R.; Applebaum, A.; Foster, H.; Little, C.; and Strom, B. 2018. Automated adversary emulation: A case for planning and acting with unknowns.

Muggleton, S., and Raedt, L. D. 1994. Inductive logic programming: Theory and methods. *JLP* 19/20:629–679.

Nebel, B.; Dimopoulos, Y.; and Koehler, J. 1997. Ignoring irrelevant facts and operators in plan generation. In *Proc. of ECP'97*, 338–350.

Pednault, E. P. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. of KR'89*, 324–331.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg,

V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; and Duchesnay, E. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12:2825–2830.

Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A sound, complete, partial order planner for ADL. In Nebel, B.; Swartout, W.; and Rich, C., eds., *Principles of Knowledge Representation and Reasoning: Proc. of the 3rd International Conference (KR-92)*, 103–114. Cambridge, MA: Morgan Kaufmann.

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

Richter, S.; Westphal, M.; and Helmert, M. 2011. LAMA 2008 and 2011 (planner abstract). In *IPC 2011 planner abstracts*, 50–54.

Ridder, B., and Fox, M. 2014. Heuristic evaluation based on lifted relaxed planning graphs. In *Proc. ICAPS'14*, 244–252.

Röger, G.; Sievers, S.; and Katz, M. 2018. Symmetry-based task reduction for relaxed reachability analysis. In *Proc. of ICAPS'18*, 208–217.

Srinivasan, A. 1999. The Aleph manual.

Torralba, Á., and Kissmann, P. 2015. Focusing on what really matters: Irrelevance pruning in merge-and-shrink. In *Proc. of SOCS'15*, 122–130.

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient symbolic search for cost-optimal planning. *Artificial Intelligence* 242:52–79.

Toyer, S.; Trevizan, F.; Thiebaux, S.; and Xie, L. 2018. Action schema networks: Generalised policies with deep learning. In McIlraith, S., and Weinberger, K., eds., *Proc. of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*. AAAI Press.

Yoon, S. W.; Fern, A.; and Givan, R. 2008. Learning control knowledge for forward search planning. *Journal of Machine Learning Research* 9:683–718.

Beyond Cost-to-go Estimates in Situated Temporal Planning

Andrew Coles,[2] Shahaf S. Shperberg,[1] Erez Karpas, [4]
Solomon E. Shimony[1] Wheeler Ruml,[3]
[1]Ben-Gurion University, Israel; [2]King's College London, UK;
[3]University of New Hampshire, USA; [4]Technion, Israel;
andrew.coles@kcl.ac.uk, shperbsh@post.bgu.ac.il, karpase@technion.ac.il
shimony@cs.bgu.ac.il, ruml@cs.unh.edu

Abstract

Heuristic search research often deals with finding algorithms for offline planning which aim to minimize the number of expanded nodes or planning time. In online planning, algorithms for real-time search or deadline-aware search have been considered before. However, in this paper, we are interested in the problem of *situated temporal planning* in which an agent's plan can depend on exogenous events in the external world, and thus it becomes important to take the passage of time into account during the planning process. Previous work on situated temporal planning has proposed simple pruning strategies, as well as complex schemes for a simplified version of the associated metareasoning problem. In this paper, we propose a simple metareasoning technique, called the crude greedy scheme, that can be applied in a situated temporal planner. Our empirical evaluation shows that the crude greedy scheme outperforms standard heuristic search based on cost-to-go estimates.

Introduction

For many years, research in heuristic search has focused on the objective of minimizing the number of nodes expanded during search (e.g (Dechter and Pearl 1985)). While this is the right objective under various scenarios, there are various scenarios where it is not. For example, if we still want an optimal plan but want to minimize search time, selective max (Domshlak, Karpas, and Markovitch 2012) or Rational Lazy A* (Karpas et al. 2018) can be used. Other work has dealt with finding a boundedly suboptimal plan as quickly as possible (Thayer et al. 2012), or with finding any solution as quickly as possible (Wilt and Ruml 2015). Departing from this paradigm even more, in motion planning the setting is that edge-cost evaluations are the most expensive operation, requiring different search algorithms (Mandalika, Salzman, and Srinivasa 2018; Haghtalab et al. 2018).

While the settings and objectives mentioned above are quite different from each other, they are all forms of offline planning. Addressing online planning raises a new set of objectives and scenarios. For example, in real-time search, an agent must interleave planning and execution, requiring still different search algorithms (Koenig and Sun 2009; Sharon, Felner, and Sturtevant 2014; Cserna, Ruml, and Frank 2017; Cserna et al. 2018). Deadline-aware search (Dionne, Thayer, and Ruml 2011) must find a plan within some deadline. The BUGSY planner (Burns, Ruml, and Do 2013) attempts to optimize the utility of a plan, which depends on both plan quality and search time.

In this paper we are concerned with a recent setting, called *situated temporal planning* (Cashmore et al. 2018). Situated temporal planning addresses a problem where planning happens online, in the presence of external temporal constraints such as deadlines. In situated temporal planning, a plan must be found quickly enough that it is possible to execute that plan after planning completes. Situated temporal planning is inspired by the planning problem a robot faces when it has to replan (Cashmore et al. 2019), but the problem statement is independent of this motivation.

The first planner to address situated temporal planning (Cashmore et al. 2018) used temporal reasoning (Dechter, Meiri, and Pearl 1991) prune search nodes for which it is provably too late to start execution. It also used estimates of remaining search time (Dionne, Thayer, and Ruml 2011) together with information from the temporal relaxed planning graph (Coles et al. 2010) to estimate whether a given search node is likely to be *timely*, meaning that it is likely to lead to a solution which will be executable when planning finishes. It also used dual open lists: one only for timely nodes, and another one for all nodes (including nodes for which it is likely too late to start execution). However, the planner still used standard heuristic search algorithms (GBFS or Weighted A*) with these open lists, while noting that this is the wrong thing to do, and leaving for future work finding the right search control rules.

Inspired by this problem, a recent paper (Shperberg et al. 2019) proposed a rational metareasoning (Russell and Wefald 1991) approach for a simplified version of the problem faced by the situated planner. The problem was simplified in several ways: first, the paper addressed a one-shot version of the metareasoning problem, and second, the paper assumed distributions on the remaining search time and on the deadline for each node are known. The paper then formulated the metareasoning problem as an MDP, with the objective of maximizing the *probability* of finding a timely plan, and showed that it is intractable. It also gave a greedy decision rule, which worked well in an empirical evaluation with various types of distributions.

In this paper, we explore using such a metareasoning approach *as an integrated part of* a situated temporal planner.

This involves addressing the two simplifications described above. The naive way of addressing the first simplification — the one-shot nature of the greedy rule — is to apply it at every expansion decision the underlying heuristic search algorithm makes, in order to choose which node from the open list to expand. The problem with this approach is that the number of nodes on the open list grows very quickly (typically exponentially), and so even a linear time metareasoning algorithm would incur too much overhead. Thus, we introduce an even simpler decision rule, which we call the *crude greedy* scheme, which does not require access to the distributions, but only to their estimated means. Additionally, the crude greedy scheme allows us to compute one number for each node, $\hat{Q}$, and expand nodes with a high $\hat{Q}$-value first. This allows us to use a regular open list, although one that is not sorted according to cost-to-go estimates, as in standard heuristic search. In fact, as we will see, cost-to-go estimates play no role in the ordering criterion at all.

An empirical evaluation on a set of problems from the Robocup Logistics League (RCLL) domain (Niemueller, Lakemeyer, and Ferrein 2015; Niemueller et al. 2016) shows that using the crude greedy scheme in the situated temporal planner (Cashmore et al. 2018) leads to a *timely* solution of significantly more problems than using standard heuristic search, even with pruning late nodes and dual open lists. Next, we briefly survey the main results of the metareasoning paper (Shperberg et al. 2019), and then describe how we derive the crude greedy decision rule, and conclude with an empirical evaluation that demonstrates its efficacy.

The metareasoning MDP and practical approximations

A model called AE2 ('allocating effort when actions expire') that assigns processing time under the simplifying assumption of n independent processes was proposed by Shperberg et al. (2019). In order to make this paper self-contained, we re-state the model and its properties below.

The AE2 Model

The AE2 model abstracts away from the search, and assumes n processes (e.g., each process can be thought of as a node on the open list) that each attempts to solve the same problem under time constraints. (For example, these may represent promising partial plans for a certain goal, implemented as nodes on the frontier of a search tree, but as discussed below the problems may be completely unrelated to planning.) There is a single computing thread or processor to run all the processes, so it must be shared. When process i terminates, it will, with probability P_i, deliver a solution or, otherwise, indicate its failure to find one. For each process, there is a deadline, defined in absolute wall clock time, by which the computation must be completed in order for any solution it finds to be valid, although that deadline may only be known to us with uncertainty. For process i, let $D_i(t)$ be the CDF over wall clock times of the random variable denoting the deadline. Note that the actual deadline for a process is only discovered with certainty when its computation is complete. This models the fact that, in planning, a dependence on an external timed event might not become clear until the final action in the plan is added. If a process terminates with a solution before its deadline, we say that it is *timely*. The processes have performance profiles described by CDFs $M_i(t)$ giving the probability that process i will terminate given an accumulated computation time on that process of t or less. Although some of the algorithms we present may work with dependent random variables, we assume in our analysis that all the variables are independent. Given the $D_i(t)$, $M_i(t)$, and P_i, the objective of AE2 is to schedule processing time over the n processes such that the probability that at least one process finds a solution before its deadline is maximized. This is the essential metareasoning problem in planning when actions expire.

The Deliberation Scheduling MDP

We now represent the AE2 problem of deliberation scheduling with uncertain deadlines as a Markov decision process. For simplicity, we initially assume that time is discrete and the smallest unit of time is 1. Allowing continuous time is more complex because one needs to define what is done if some time-slice is allocated to a process i, and that process terminates before the end of the time-slice. Discretization avoids this complication.

We can now define our deliberation scheduling problem as an the following MDP, with distributions represented by their discrete probability function (pmf). Denote $m_i(t) = M_i(t) - M_i(t-1)$, the probability that process i completes after exactly t time units of computation time, and $d_i(t) = D_i(t) - D_i(t-1)$, the probability that the deadline for process i is exactly at time t. Without loss of generality, we can assume that $P_i = 1$: otherwise modify the deadline distribution for process i to have $d_i(-1) = 1 - P_i$, simulating failure of the process to find a solution at all with probability $1 - P_i$, and multiply all other $d_i(t)$ by P_i. This simplified problem we call SEA2. We formalize the SEA2 MDP as an indefinite duration MDP with terminal states, where we keep track of time as part of the state. (An alternate definition would be as a finite-horizon MDP, given a finite value d for the last possible deadline.)

The actions in the MDP are: assign the next time unit to process i, denoted by a_i with $i \in [1, n]$. We allow action a_i only if process i has not already failed.

The state variables are the wall clock time T and one state variable T_i for each process, with domain $\mathcal{N} \cup \{F\}$. T_i denotes the cumulative time assigned to each process i until the current state, or that the process has completed computation and resulted in failure to find a solution within the deadline. We also have special terminal states SUCCESS and FAIL. Thus the state space is:

$$\mathcal{S} = (dom(T) \times \bigtimes_{1 \leqslant i \leqslant n} dom(T_i)) \cup \{\text{SUCCESS, FAIL}\}$$

The initial state is $T = 0$ and $T_i = 0$ for all $1 \leqslant i \leqslant n$.

The transition distribution is determined by which process i has last been scheduled (the action a_i), and the M_i and D_i distributions. If all processes fail, transition into FAIL with probability 1. If some process is successful, transition into SUCCESS with probability 1. More precisely:

- The current time T is always incremented by 1.
- Accumulated computation time is preserved, i.e. for action a_i, $T_j(t+1) = T_j(t)$ for all processes $j \neq i$.
- $T_i(t) = F$ always leads to $T_i(t+1) = F$.
- For action a_i (assign time to process i), the probability that process i's computation is complete given that it has not previously completed is $P(C_i) = \frac{m_i(T_i+1)}{1-M_i(T_i)}$. If completion occurs, the respective deadline will be met with probability $1 - D_i(T_i)$. Therefore, transition probabilities are: with probability $1 - P(C_i)$ set $T_i(t+1) = T_i(t) + 1$, with probability $P(C_i)D_i(T_i)$ set $T_i(t+1) = F$ (process i failed to meet its deadline), and otherwise (probability $P(C_i)(1 - D_i(T_i))$ transition into SUCCESS (the value of T_i in this case is 'don't care').
- If $T_i(t+1) = F$ for all i, transition into FAIL.

The reward function is 0 for all states, except SUCCESS, which has a reward of 1.

Solving the AE2 Model

It was shown in Shperberg et al. (2019) that solving the AE2 MDP is NP-hard, and it was conjectured to be even harder (possibly even PSPACE-complete, like similar MDPs). On the other hand, under the restriction of known deadlines and a special condition of diminishing returns (non-decreasing logarithm of probability of failure) that an optimal schedule can be found in polynomial time. However, neither known deadlines nor diminishing returns strictly hold in practice in planning processes. Still, the algorithm for diminishing returns provided insights that were used to create an appropriate greedy scheme. The greedy scheme, briefly repeated below, is relatively easy to compute and achieved good results empirically.

Define $m_i(t) = M_i(t) - M_i(t-1)$, the probability that process i completes after exactly t time units of computation time. Under an allocation $A_i = (0, t)$ in which all processing time starting from time 0 until time t is allocated to process i, the success distribution for process i is:

$$f_i(t) = PS_i(A_i = (0,t)) = P_i \sum_{t'=0}^{t} m_i(t')(1 - D_i(t')) \quad (1)$$

Define the *most effective computation time* for process i under this assumption to be:

$$e_i = \operatorname*{argmin}_t \frac{log(1 - f_i(t))}{t} \quad (2)$$

The latter is justified by observing that the term $-log(1 - f_i(t))$ behaves like utility, as it is monotonically increasing with the probability of finding a timely plan in process i; and on the other hand it behaves additively with the terms for other processes. That is, if we could start all processes at time 0 and run them for time t, and if all the random variables were jointly independent, then indeed maximizing the sum of the $-log(1 - f_i(t))$ terms results in maximum probability of a timely plan.

However, since not all processes can start at time 0, the intuition from the diminishing returns optimization is thus to prefer the process i that has the best utility per time unit. i.e. such that $-log(1 - f_i(t))/(e_i)$ is greatest. Still, allocating time now to process i delays other processes, so it is also important to allocate the time now to a process that has a deadline as early as possible, as this is most critical. Shperberg et al. (2019) therefore suggested the following greedy algorithm: Whenever assigning computation time, allocate t_d units of computation time to process i that maximizes:

$$Q(i) = \frac{\alpha}{E[D_i]} - \frac{log(1 - f_i(e_i))}{e_i} \quad (3)$$

where α and t_d are positive empirically determined parameters, and $E[D_i]$ is the expectation of the random variable that has the CDF D_i, which we use as a proxy for 'deadline of process i'. (This is a slight abuse of notation in the interest of conciseness, as $E[D_i]$ could be taken to mean the expectation of the CDF, which is *not* what we want here.) The α parameter trades off between preferring earlier expected deadlines (large α) and better performance slopes (small α).

Improved Greedy Scheme

Using the proxy $E[D_i]$ in the value $Q(i)$ is reasonable, but somewhat ad-hoc. It also encounters problems if $E[D_i]$ is zero or even near-zero. A more disciplined scheme can indeed use the utility per time unit as in $Q(i)$, but the first term should be better justified theoretically. The reason for including the deadline in $Q(i)$ is in order to give preference to processes with an early deadline, because deferring their processing may cause them to be unable to complete before their deadline (even if they *would* have been timely had they been scheduled for processing immediately). Therefore, instead of the first term it makes sense to provide a measure of the "utility damage" to a process i due to delaying its processing start time from time 0 to time t_d. This can be computed exactly, as follows. Define a 'generalized' f'_i, the probability of process i finding a timely plan given a contiguous computation time t starting at time t_d, as follows:

$$f'_i(t, t_d) = PS_i(A_i = (t_d, t)) = P_i \sum_{t'=0}^{t} m_i(t')(1 - D_i(t' + t_d)) \quad (4)$$

Note that this is the same as f, except that processing starts at time t_d, which is the same as saying that the deadline distribution is advanced by t_d (and indeed, $f_i(t) = f'_i(t, 0)$).

Assuming that the time we wish to assign to process i is e_i, before the delay we can achieve a utility of: $-log(1 - f_i(e_i))$, and after delay of t_d can achieve $-log(1 - f'_i(e_i, t_d))$. The difference between the former and the latter values is the 'damage' caused by the delay. Thus, our improved greedy scheme is to assign t_d time units to the process that maximizes:

$$Q'(i) = \alpha[log(1 - f'_i(e_i, t_d)) - log(1 - f_i(e_i))] - \frac{log(1 - f_i(e_i))}{e_i} \quad (5)$$

Observe that the first term is proportional to the logarithm of:

$$\frac{1 - f'_i(e_i, t_d)}{1 - f_i(e_i)} \quad (6)$$

Integrating the greedy scheme into a planner

In order to actually use the greedy scheme in a planner, several issues must be handled. Foremost is the issue of obtaining the distributions, which is non-trivial. Second, although the greedy scheme is quite efficient, it is not quite efficient enough for making decisions about node expansions, which must be done in essentially negligible time. Hence, we consider a crude version of the greedy scheme below.

Crude version of the greedy scheme

		Crude Greedy with $\alpha =$					
Id	h	-10^4	-1	0	0.1	1	10^4
1	3.61	0.45	0.47	0.45	0.85	0.45	0.59
2	13.45	1.91	1.81	1.78	2.52	1.77	x
3	x	1.75	1.74	1.61	2.24	1.65	5.14
4	x	1.03	1.04	1	1.41	1.02	1.19
5	9.42	0.59	0.51	0.57	0.73	0.49	0.4
6	-	-	-	-	-	-	-
7	x	10.89	9.74	9.63	17.55	9.97	x
8	x	x	3.57	3.77	5.98	3.88	2.04
9	x	2.36	2.46	2.5	3.16	2.4	2.27
10	0.66	0.37	0.37	0.38	0.49	0.38	0.36
11	0.28	0.24	0.24	0.25	0.31	0.27	9.05
12	0.31	0.25	0.3	0.24	0.34	0.24	0.17
13	0.9	0.44	0.45	0.46	0.55	0.45	0.43
14	11.88	1.55	1.64	1.6	2.11	1.59	1.19
15	1.54	x	x	x	x	x	0.59
16	x	2.33	2.27	2.29	3.56	2.29	1.71
17	x	1.59	1.55	1.54	2.52	1.59	3.93
18	x	2.27	2.27	2.35	4.34	2.29	x
19	x	1.61	1.6	1.62	2.73	1.6	6.6
20	-	-	-	-	-	-	-
21	x	x	x	x	x	x	1.71
22	0.76	0.43	0.43	0.41	0.42	0.48	1.37
23	x	2.06	2.33	1.99	2.04	2.36	1.78
24	14.67	1.55	1.61	1.49	1.63	1.92	0.52
25	0.57	0.31	0.32	0.28	0.28	0.35	0.46
26	4	x	x	x	x	x	x
27	x	1.92	1.88	1.75	1.72	1.82	x
28	x	0.52	0.74	0.49	0.47	0.47	0.48
29	50.03	x	1.21	1.13	1.13	1.23	0.56
30	-	-	-	-	-	-	-
31	2.48	x	x	x	x	x	0.96
32	x	1.74	1.79	1.56	1.91	1.74	1.59
33	x	x	x	x	x	x	x
34	3.37	0.54	0.63	0.54	0.27	0.59	0.26
35	2.73	x	x	x	x	0.34	0.23
36	10.18	0.75	0.76	0.62	0.57	0.87	0.71
37	-	-	-	-	-	-	-
38	-	-	-	-	-	-	-
39	1.09	0.54	0.65	0.48	0.49	0.52	0.49
40	-	-	-	-	-	-	-
41	1.28	0.27	0.28	0.27	0.52	0.28	0.24
42	1.32	x	x	x	x	x	0.42
SOLVED	21	27	29	29	29	**30**	**30**

Table 1: Planning Time on RCLL Instances

Consider Equation 3 defining $Q(i)$. The estimate for $E[D_i]$ in the first term can use any current estimate of the deadline time. For the second term in $Q(i)$, we can approximate e_i by the expected time to return a solution. We use estimate both of these quantities as described by Cashmore et al. (2018). We now briefly review these estimates, but refer the interested reader to the original paper.

To estimate the current deadline $E[D_i]$, we use the temporal relaxed planning graph (TRPG) (Coles et al. 2010). Specifically, we compute the *slack* of the chosen relaxed plan, that is, how much we can delay execution of the entire plan (the actions leading to the current node together with the actions in the relaxed plan). Note that, because the relaxed plan is not guaranteed to be optimal, this is not necessarily an admissible estimate.

To estimate the remaining search time e_i, we use an idea from Deadline Aware Search (Dionne, Thayer, and Ruml 2011). We estimate the 'distance from solution' (i.e. estimation of number of expansions from the current node, also based on the relaxed), and divide it by the 'progress rate' (i.e. the reciprocal of the time difference between the time a node is expanded and the time its parent was expanded, averaged over multiple nodes).

The numerator $\log(1 - f_i(e_i))$ is more problematic, as it requires f_i, which uses the complete distribution. Note that this term is negative, and we want it to be as large as possible in absolute value. The simplest crude approximation is a constant $\log(1 - f_i(e_i))$, but that is an oversimplification. Note that if the most effective computation time e_i is greater than $E[D_i]$ then in fact we are not likely to find a solution in time in process i. For simplicity, we thus use $-log(1 - f_i(e_i)) \approx max(0, \beta(E[D_i] - e_i))$ for some parameter β as a first approximation. The idea here is that $E[D_i] - e_i$ is an estimate of the *slack* (spare time) we have in completing the computation before the deadline. We are then assuming that the negative logarithm of the probability of not completing in time is approximately proportional to the slack. This slack is also already estimated by the situated temporal planner, based on the partial plan to the current node and the temporal relaxed planning graph from it (Cashmore et al. 2018).

Note that once we plug this into Equation 3, the β can be absorbed into the α parameter. An additional issue is that in a planner, since $E[D_i]$ is relative to the time now, this value keeps decreasing and may approach 0. This may cause the $\frac{\alpha}{E[D_i]}$ term to grow without bound. To fix this, we bound the denominator away from 0 to the time t_{10}, the time required for 10 node expansions.

In summary, for our crude greedy approach, we expand next the node with the highest value of

$$\hat{Q}(i) = \frac{\max(0, E[D_i] - e_i)}{e_i} + \frac{\alpha}{\max(E[D_i], t_{10})} \quad (7)$$

This crude version of the greedy scheme has two advantages: it does not require the complete distributions D_i and M_i, and is more computationally efficient as it does not have to compute the summation in the equation for f_i. This comes at the cost of a potential oversimplification that may cause schedule quality to excessively degrade. An additional problem is that the original greedy scheme itself using the $E[D_i]$ was only a first-order approximation, and in fact distributions can be devised where it fails badly.

Empirical Results

To evaluate the crude greedy scheme, we implemented it on top of the situated temporal planner of Cashmore et al. (2018), which itself is implemented on top of OPTIC (Benton, Coles, and Coles 2012). We ran the planner using the crude greedy scheme, with different values of α, and compared it to the original situated temporal planner, which sorts its open lists based on cost-to-go estimates (denoted h below). Both planners used exactly the same pruning method for nodes which are guaranteed to be too late, and the same dual open list mechanism for preferring nodes which are likely to be timely.

We compared the results of the different planners on instances of the Robocup Logistic League Challenge (Niemueller, Lakemeyer, and Ferrein 2015; Niemueller et al. 2016), a domain that involves robots moving workpieces between different workstations. The goal is to manufacture and deliver an order within some time window, and thus situated temporal planning is very natural here. Table 1 shows the planning time for the baseline planner (h) and the planner using the crude greedy scheme with different values of α. In the table, 'x' means 'failed to find a plan in time to satisfy the deadline(s)'. As these results show, the crude greedy scheme solves significantly more problems than the baseline for any value of α. This provides support for a metareasoning approach to allocating search effort in situated planning. It also suggests that, for situated temporal planning, cost-to-go estimates are not the right primary source of heuristic guidance.

Conclusion

In this paper, we have provided the first practical metareasoning approach for situated temporal planning. We showed empirically that this approach outperforms standard heuristic search based on cost-to-go estimates. Nevertheless, the temporal relaxed planning graph (Coles et al. 2010) serves an important purpose here, allowing us to estimate both remaining planning time and the deadline for a node. Thus, we believe our results suggest that cost-to-go estimates are not as important for situated temporal planning as they are for minimizing the number of expanded nodes or planning time as in classical heuristic search.

The metareasoning scheme we provided is a crude version of the greedy scheme of Shperberg et al. (2019). We introduced approximations in order to make the metareasoning sufficiently fast and in order to utilize only readily available information generated during the search. We also proposed a more refined and better theoretically justified version of the algorithm ('improved greedy'), but making the improved version applicable in the planner is a non-trivial challenge that forms part of our future research.

Ongoing Work: Crude version of the improved greedy scheme

The improved greedy scheme is better justified, but has an additional term where we need the complete distribution ($f'(t, t_d)$ is needed, rather than just the expectation $E[D_i]$). We would like to replace this distribution with a small number of parameters than can be easier to obtain. Basically the same considerations apply here as well, except that the the term involving f'_i requires access to the full distributions m_i, D_i. Given specific distribution types, it may be possible to compute this term as a function of $E[D_i]$ and e_i. However, this part of the work is still in progress and at present we are not sure what parameters we can obtain during the search that would support the improved scheme.

Acknowledgements

Partially supported by ISF grant #844/17, and by the Frankel center for CS at BGU. Project also funded by the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement No. 821988 (ADE).

References

Benton, J.; Coles, A. J.; and Coles, A. 2012. Temporal planning with preferences and time-dependent continuous costs. In *ICAPS*.

Burns, E.; Ruml, W.; and Do, M. B. 2013. Heuristic search when time matters. *J. Artif. Intell. Res.* 47:697–740.

Cashmore, M.; Coles, A.; Cserna, B.; Karpas, E.; Magazzeni, D.; and Ruml, W. 2018. Temporal planning while the clock ticks. In *ICAPS*, 39–46.

Cashmore, M.; Coles, A.; Cserna, B.; Karpas, E.; Magazzeni, D.; and Ruml, W. 2019. Replanning for situated robots. In *ICAPS*.

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-chaining partial-order planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS)*, 42–49.

Cserna, B.; Doyle, W. J.; Ramsdell, J. S.; and Ruml, W. 2018. Avoiding dead ends in real-time heuristic search. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, 1306–1313.

Cserna, B.; Ruml, W.; and Frank, J. 2017. Planning time to think: Metareasoning for on-line planning with durative actions. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017.*, 56–60.

Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of a*. *J. ACM* 32(3):505–536.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49(1-3):61–95.

Dionne, A. J.; Thayer, J. T.; and Ruml, W. 2011. Deadline-aware search using on-line measures of behavior. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*.

Domshlak, C.; Karpas, E.; and Markovitch, S. 2012. Online speedup learning for optimal planning. *J. Artif. Intell. Res.* 44:709–755.

Haghtalab, N.; Mackenzie, S.; Procaccia, A. D.; Salzman, O.; and Srinivasa, S. S. 2018. The provable virtue of laziness in motion planning. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018.*, 106–113.

Karpas, E.; Betzalel, O.; Shimony, S. E.; Tolpin, D.; and Felner, A. 2018. Rational deployment of multiple heuristics in optimal state-space search. *Artif. Intell.* 256:181–210.

Koenig, S., and Sun, X. 2009. Comparing real-time and incremental heuristic search for real-time situated agents. *Autonomous Agents and Multi-Agent Systems* 18(3):313–341.

Mandalika, A.; Salzman, O.; and Srinivasa, S. 2018. Lazy receding horizon a* for efficient path planning in graphs with expensive-to-evaluate edges. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018.*, 476–484.

Niemueller, T.; Karpas, E.; Vaquero, T.; and Timmons, E. 2016. Planning Competition for Logistics Robots in Simulation. In *ICAPS Workshop on Planning and Robotics (PlanRob)*.

Niemueller, T.; Lakemeyer, G.; and Ferrein, A. 2015. The RoboCup Logistics League as a Benchmark for Planning in Robotics. In *WS on Planning and Robotics (PlanRob) at Int. Conf. on Aut. Planning and Scheduling (ICAPS)*.

Russell, S. J., and Wefald, E. 1991. Principles of metareasoning. *Artificial Intelligence* 49(1-3):361–395.

Sharon, G.; Felner, A.; and Sturtevant, N. R. 2014. Exponential deepening a* for real-time agent-centered search. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, 871–877.

Shperberg, S. S.; Coles, A.; Cserna, B.; Karpas, E.; Ruml, W.; and Shimony, S. E. 2019. Allocating planning effort when actions expire. In *Proceedings of the The Thirty-Third AAAI Conference on Artificial Intelligence (AAAI)*. (to appear).

Thayer, J. T.; Stern, R.; Felner, A.; and Ruml, W. 2012. Faster bounded-cost search using inadmissible estimates. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling, (ICAPS)*.

Wilt, C. M., and Ruml, W. 2015. Speedy versus greedy search. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 4331–4338.