Pattern Selection Strategies for Pattern Databases in Probabilistic Planning

Thorsten Klößner[1], **Marcel Steinmetz**[1], **Álvaro Torralba**[2], **Jörg Hoffmann**[1]

[1]Saarland University, Saarland Informatics Campus, Germany
[2]Aalborg University, Department of Computer Science, Denmark
{kloessner, hoffmann, steinmetz}@cs.uni-saarland.de, alto@cs.aau.dk

Abstract

Recently, pattern databases have been extended to probabilistic planning, to derive heuristics for the objectives of goal probability maximization and expected cost minimization. While this approach yields both theoretical and practical advantages over techniques relying on determinization, the problem of selecting the patterns in the first place has only been scantily addressed as yet, through a method that systematically enumerates patterns up to a fixed size. Here we close this gap, extending pattern generation techniques known from classical planning to the probabilistic case. We consider hill-climbing as well as counter-example guided abstraction refinement (CEGAR) approaches, and show how they need to be adapted to obtain desired properties such as convergence to the perfect value function in the limit. Our experiments show substantial improvements over systematic pattern generation and the previous state of the art.

Introduction

In probabilistic planning as we address here, actions specify a probability distribution over possible outcomes. Heuristic search is a prominent approach to solve such problems (e. g. Hansen and Zilberstein (2001); Bonet and Geffner (2003); Trevizan et al. (2017)). Yet the arsenal of heuristics actually taking the probabilities into account has long been limited. Most works rely on a determinization of the problem (Little and Thiébaux 2006; Steinmetz, Hoffmann, and Buffet 2016b,a; Trevizan, Teichteil-Königsbuch, and Thiébaux 2017; Klauck et al. 2020). Other works addressed particular cases, namely probabilistic conformant planning (Bryce, Kambhampati, and Smith 2006; Domshlak and Hoffmann 2007) and finding a maximum-likelihood sequential plan (Keyder and Geffner 2008; E-Martín, Rodríguez-Moreno, and Smith 2014). Only a single line of works in the past devised admissible heuristics for stochastic shortest path problems (SSP) which are not based on determinization (Trevizan, Thiébaux, and Haslum 2017). Here we follow up on the work by Klößner et al. (2021); Klößner and Hoffmann (2021) (henceforth: *Klößner21*) who address this deficiency through the extension of pattern database (PDB) heuristics to the probabilistic setting. Our contribution is the design of pattern selection strategies in this context.

Like Klößner21, we consider two variations of probabilistic planning, namely SSPs, where expected cost-to-goal must be minimized, as well as MaxProb, where the probability of reaching the goal must be maximized. Various heuristic search algorithms exist for both settings (Hansen and Zilberstein 2001; Bonet and Geffner 2003; Trevizan et al. 2017). Heuristic search algorithms for MaxProb (SSP problems) guarantee finding an optimal policy when provided with an admissible heuristic, that provides upper (lower) bounds for the optimal goal-probability (expected cost-to-goal) for a state.

Pattern Database heuristics are an important subclass of abstraction heuristics in classical planning (Korf 1997; Culberson and Schaeffer 1998; Edelkamp 2001; Holte et al. 2006; Haslum et al. 2007; Pommerening, Röger, and Helmert 2013). These heuristics consider several projections of the problem, in which only a subset of variables (a pattern) is considered and other variables are ignored. Such a PDB heuristic is constructed in two steps: (i) Generate a collection of patterns and (ii) for each pattern, construct a lookup table containing all abstract state costs of the projection (the pattern database). Each PDB provides a lower-bounding heuristic that can estimate the cost-to-goal of a state in classical planning, by looking up the cost of the corresponding abstract state. Estimates of a collection of PDBs can be combined either by simply taking the maximal estimate among all PDBs, or using more sophisticated methods such as additivity constraints under which the sum over patterns is admissible (Haslum et al. 2007).

By using PDB heuristics on the all-outcomes determinization (pretending that one can choose the action outcome), lower-bounding (respectively upper-bounding) heuristics for SSP problems and MaxProb can be obtained. However, as Klößner21 demonstrated, projections can also be formulated on probabilistic problems in a factored representation, and these probabilistic projections always provide better estimates than the aforementioned approach. Klößner21 develop probabilistic variants of PDB heuristics, in which the lookup tables are constructed from the probabilistic projection. But they do not address the important question of how to obtain the pattern collection in the first place (instead they use a simple enumerative strategy in their experiments).

Here we fill that gap, for both the SSP and the MaxProb setting. We adapt two popular strategies known from classical planning: Counterexample guided abstraction refinement (CEGAR) (Rovner, Sievers, and Helmert 2019) and hill-climbing search in the space of pattern collections (Haslum et al. 2007), which we only pursue for SSP problems. We analyze how these methods need to be extended for the probabilistic setting. In particular, we show that substantial extensions need to be made to CEGAR, as a naïve extension loses the crucial property of convergence to the perfect value function, i. e., the ability to eventually address all sources of information loss.

For our empirical evaluation, we experiment with our new pattern selection algorithms on a large set of PPDDL benchmarks from the International Probabilistic Planning Competition (IPPC), targeting both MaxProb and SSP problems. Our modifications show a considerable improvement over current state-of-the art heuristics, represented by MaxProb-PDBs with systematic pattern generation in MaxProb (Klößner et al. 2021), as well as the LP-based occupation measure heuristic h^{roc} (Trevizan, Thiébaux, and Haslum 2017) in the SSP setting. While our new CEGAR generator achieves good results especially in MaxProb, our hill-climbing approach achieves best results for SSP problems.

Background

Probabilistic Planning Tasks We address probabilistic planning tasks formulated in PPDDL (Younes et al. 2005). Internally, the planner represents a problem as a probabilistic SAS$^+$ task (Trevizan, Thiébaux, and Haslum 2017). A probabilistic SAS$^+$ task is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, s_\mathcal{I}, \mathcal{G} \rangle$. $\mathcal{V}$ denotes the *variables*, where each v is associated with a finite domain $\mathcal{D}_v$ of at least two values. A *partial state* is a partial function $s : \mathcal{V} \mapsto \bigcup_{v \in \mathcal{V}} \mathcal{D}_v \cup \{\bot\}$ with $s(v) \in \mathcal{D}_v \cup \{\bot\}$. If $s(v) = \bot$, we say s is undefined on v. We denote by $\mathcal{V}(s)$ the set of all variables for which s is defined. s is a *state* if $\mathcal{V}(s) = \mathcal{V}$. For a set of variables $P \subseteq \mathcal{V}$ and partial state s, we denote by $s[P]$ the *projection of s onto P* and define the set $S[P] := \{s[P] \mid s \in S\}$. We say s *subsumes* t, written $t \subseteq s$, if $s(v) = t(v)$ for all $v \in \mathcal{V}(t)$. The *application* of partial state e onto partial state s is defined by $\text{appl}(s, e)(v) = e(v)$ if e is defined on v, and $s(v)$ otherwise. The *initial state* $s_\mathcal{I}$ is a state. The *goal* $\mathcal{G}$ is a partial state. $\mathcal{A}$ is the set of *actions*. An action a specifies its *precondition* pre_a, and a probability distribution $\Pr_a$ over effects, where an effect is a partial state. The possible effects of a are denoted $\text{Eff}(a) := \{e \mid \Pr_a(e) > 0\}$.

MDPs and Optimization Objectives As the baseline model of the probabilistic system, we assume a goal-oriented Markov Decision Process (MDP) with infinite horizon as a 5-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_\mathcal{I}, \mathcal{S}_\mathcal{G} \rangle$. Here, $\mathcal{S}$ is a finite, non-empty set of states, $\mathcal{A}$ is a finite, non-empty set of actions, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a transition probability function, $s_\mathcal{I} \in \mathcal{S}$ is the initial state and $\mathcal{S}_\mathcal{G} \subseteq \mathcal{S}$ is a set of goal states. We require that for each state-action pair (s, a), either $\sum_{t \in \mathcal{S}} \mathcal{T}(s, a, t) = 1$, in which case a is *enabled* in s, or $\mathcal{T}(s, a, t) = 0$ for all t in which case a is *disabled* in s. We write $\mathcal{A}(s)$ for the set of actions that are enabled in s. We assume $\mathcal{A}(s) \neq \emptyset$ for simplicity, which can be enforced by introducing an artificial self-loop action. The task $\langle \mathcal{V}, \mathcal{A}, s_\mathcal{I}, \mathcal{G} \rangle$ induces the MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, s_\mathcal{I}, \mathcal{S}_\mathcal{G} \rangle$ in which $\mathcal{S}$ are the states of Π and $\mathcal{T}$ is defined by $\mathcal{T}(s, a, t) := 0$ if $pre_a \not\subseteq s$ and $\mathcal{T}(s, a, t) := \sum_{e.\, \text{appl}(s,e)=t} \Pr_a(e)$ otherwise. The goal states are $\mathcal{S}_\mathcal{G} = \{s \in \mathcal{S} \mid \mathcal{G} \subseteq s\}$.

A policy is a mapping $\pi : \mathcal{S} \to \mathcal{A}$ with $\pi(s) \in \mathcal{A}(s)$ for every state $s \in \mathcal{S}$. The quality of a policy depends on the optimization objective that is considered. Stochastic Shortest Path Problems (SSPs, Bertsekas (1995)) additionally assume a non-negative action cost function $c : \mathcal{A} \to \mathbb{R}_0^+$ and consider the optimization objective of minimizing the expected cost until the goal is reached. We henceforth call this objective the *SSP objective*. Additionally, we consider the *MaxProb objective*, which considers those policies as optimal which maximize the probability to reach the goal.

We associate with every policy π the MaxProb policy value function $V^\pi : \mathcal{S} \to [0, 1]$ which assigns each state s the probability of reaching the goal when starting in s and following policy π. Formally, we define V^π as the (pointwise) *least* solution x of equation system (1).

$$x(s) = \begin{cases} 1 & s \in \mathcal{S}_\mathcal{G} \\ \sum_{t \in \mathcal{S}} \mathcal{T}(s, \pi(s), t) x(t) & s \notin \mathcal{S}_\mathcal{G} \end{cases} \tag{1}$$

We say that a policy π is proper if and only if $V^\pi(s) = 1$ for every state $s \in S$, i.e. the goal will be reached surely when starting in any state of the problem and following π.

On the other hand, the SSP objective makes two additional assumptions: (i) there exists at least one proper policy and (ii) every improper policy accumulates a cost of ∞. Under these conditions, the SSP policy value function $J^\pi : S \to \mathbb{R}_0^+$ associates a state s with the cost that would accumulate in expectation until the goal is reached, when following π. J^π is defined only for proper policies π as the *unique* solution x of equation system (2).

$$x(s) = \begin{cases} 0 & s \in \mathcal{S}_\mathcal{G} \\ c(\pi(s)) + \sum_{t \in \mathcal{S}} \mathcal{T}(s, \pi(s), t) x(t) & s \notin \mathcal{S}_\mathcal{G} \end{cases} \tag{2}$$

For both objectives, the goal is to compute an optimal policy for the initial state. To this end, the optimal MaxProb value function V^* and the optimal SSP value function J^* are defined by $V^* := \max_\pi V^\pi$ and $J^* := \min_{\pi \text{ proper}} J^\pi$, where pointwise ordering is imposed on the value functions. The maximum always exists (Puterman 1994), while the minimum exists under the SSP assumptions stated above. A policy $\pi^\star$ is MaxProb-optimal for s if $V^{\pi^\star}(s) = V^*(s)$ and SSP-optimal if $J^{\pi^\star}(s) = J^*(s)$. A policy is optimal if it is optimal for all states. Furthermore, a policy π is MaxProb-greedy with respect to the value function V if

$$\pi(s) \in \operatorname*{arg\,max}_{a \in \mathcal{A}(s)} \left\{ \sum_{t \in \mathcal{S}} \mathcal{T}(s, a, t) V(t) \right\}$$

and SSP-greedy with respect to V if

$$\pi(s) \in \operatorname*{arg\,min}_{a \in \mathcal{A}(s)} \left\{ c(a) + \sum_{t \in \mathcal{S}} \mathcal{T}(s, a, t) V(t) \right\}$$

For SSP problems, the optimal policies are exactly the greedy policies with respect to J^*, so it suffices to compute J^*. For MaxProb, not every greedy policy with respect to V^* is necessarily optimal, although the other direction still holds. Here, actions must be chosen in a way that does not introduce cycles. Nevertheless, an optimal policy can be extracted from the optimal value function in both cases.

Since we are only interested in a policy that is optimal when starting in the initial state $s_\mathcal{I}$, it also suffices to compute a *partial* policy $\hat{\pi} : \mathcal{S} \to \mathcal{A} \cup \{\bot\}$ which is restricted to the states it may actually reach. To this end, we say $\hat{\pi}$ is *closed* for state s_0 if we may never reach a state s for which $\hat{\pi}(s) = \bot$ when starting in s_0. In this case $V^{\hat{\pi}}$ and $J^{\hat{\pi}}$ are naturally defined on these reachable states by restricting the defining equation systems to these states only. There exist many heuristic search algorithms for the SSP and MaxProb setting which can compute an optimal partial policy closed for the initial state without relying on the exhaustive expansion of the entire state space. While the exact inner workings of these algorithms are not relevant in this paper, it suffices to know that these algorithms guarantee to find optimal policies with respect to the MaxProb and SSP objectives when provided with a heuristic function h with the following characteristics. In the MaxProb setting, h must be an upper bound on the optimal value function ($h(s) \geq V^*(s)$), while for SSPs the heuristic must be a lower bound ($h(s) \leq J^*(s)$). In both cases, such a heuristic is denoted *admissible* for the respective objective.

Pattern Databases for MaxProb and SSPs Klößner21 introduce a probabilistic variant of projection heuristics which is formulated on MDPs. Given a pattern $P \subseteq \mathcal{V}$, they define this projection MDP as $\langle \mathcal{S}[P], \mathcal{A}, s_\mathcal{I}[P], \mathcal{T}_P, \mathcal{S}_\mathcal{G}[P] \rangle$ where the transition function is defined as

$$\mathcal{T}_P(\sigma, a, \tau) := \begin{cases} \sum\limits_{e.\, \mathrm{appl}(\sigma, e[v]) = \tau} \Pr_a(e) & \sigma \subseteq pre_a[P] \\ 0 & \text{otherwise} \end{cases}$$

and the cost function for the SSP objective is inherited.

Denoting with V_P^* and J_P^* the optimal value functions of this MDP with respect to MaxProb and SSP problems, they show that the heuristics $h_P^{\text{MaxProb}}(s) = V_P^*(s[P])$ and $h_P^{\text{SSP}}(s) = J_P^*(s[P])$ are admissible for the respective objective. Also, these heuristics dominate their determinization-based counterparts, which utilize the classical projection on the determinization to detect dead-ends in MaxProb, while considering the cost-to-goal in the SSP setting.

Moreover, Klößner21 derive efficiently verifiable constraints under which it is possible to combine multiple PDB heuristics by addition (SSP) and multiplication (MaxProb) of their individual estimates while remaining admissible. The exact constraints are not of particular interest to us, as our constructions are mostly agnostic to the particular type of additivity/multiplicativity constraint. Throughout this paper, we will assume an arbitrary combination strategy $\mathcal{Q}$ which maps any pattern collection C to a set of additive/multiplicative subcollections $\mathcal{Q}(C) \subseteq \mathcal{P}(C)$ depending on the setting. The additive/multiplicative pattern collection heuristic h_C for any family of heuristics h_P defined on patterns P is then generally defined by either

$$h_C(s) = \max_{C' \in \mathcal{Q}(C)} \left\{ \sum_{P \in C'} h_P(s) \right\}$$

in the context of SSP problems, or

$$h_C(s) = \min_{C' \in \mathcal{Q}(C)} \left\{ \prod_{P \in C'} h_P(s) \right\}$$

in the context of MaxProb.

Pattern Generation by Counterexample Guided Abstraction Refinement

Counterexample guided abstraction refinement (CEGAR) is an incremental technique to compute abstractions which is popular in both model checking and automated planning (e.g. Clarke et al. (2003), Hermanns, Wachter, and Zhang (2008), Seipp (2012), Seipp and Helmert (2018)). In its core, CEGAR iteratively refines an abstraction of the state space until the abstraction satisfies a property of interest or a resource limit is reached. Especially interesting for us is its use in a generation algorithm for pattern collections in classical planning by Rovner, Sievers, and Helmert (2019), using projection as the underlying abstraction type.

In this setting, the general refinement procedure for a single projection with respect to pattern P is summarized as follows. One or more optimal abstract plans of the projection are computed and it is checked whether one of them constitutes an optimal plan for the original task. First, it is checked whether the abstract plan is executable in the original problem. If the execution fails, this is due to a precondition on some variable $v \notin P$ which is not satisfied at some point during plan execution (a *precondition flaw*). However, even if the abstract plan is executable, the sequence of actions may not lead to a goal state, as the goal value of some variable $v \notin P$ may not have been reached (a *goal flaw*). If no abstract plan solves the original problem, the projection is refined by inserting one or more such variables, otherwise the found plan constitutes an optimal plan for the original problem and the procedure terminates by reporting the solution. In theory, this procedure is complete in the sense that an optimal plan will eventually be found when repeating this procedure, as the number of variables is limited.

Determinization-Based CEGAR

By leveraging the all-outcomes determinization of the planning task, we can theoretically use any instantiation of this framework to generate patterns for probabilistic problems. Unfortunately, the aforementioned completeness guarantee does not transfer to probabilistic setting since we are checking for plans instead of policies. Also, the CEGAR refinement procedure may terminate extremely early by finding a plan in the determinization of the task.

As an example, we consider the IPPC domain *triangle-tireworld.* This domain is explicitly designed to provoke the worst-case behaviour of determinization-based planners. A problem instance of this domain is depicted in Figure 1.

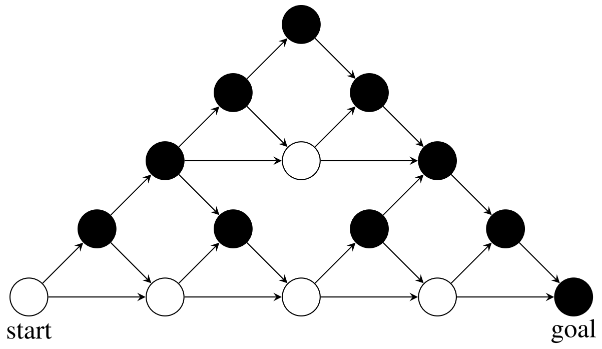

Figure 1: An instance of the triangle-tireworld domain. Locations with spare tires are black.

In this domain, a vehicle has to traverse a triangle-shaped roadmap to arrive at a destination. The vehicle can *move* between two adjacent locations. When moving, the vehicle may get a flat tire with some positive probability, after which it can no longer move anywhere. However, spare tires are scattered across the map which may be *loaded* and used to *change* the broken tire. In particular, there are spare tires on each of the outermost points on the triangle. At most one tire may be loaded at a time. Initially no spare tire is loaded.

In the example, the unique optimal policy for the MaxProb and SSP objective uses the outermost path, as this is the only path on which a spare tire is always available if needed. However, an optimal plan in the determinization would simply take the shortest possible path to the goal, as we can always optimistically choose the effect which does not lead to a flat tire. This plan only modifies the vehicle location variable and otherwise only depends on variable preconditions that are initially satisfied. Consequently, CEGAR terminates immediately when starting with the single vehicle variable, as the induced optimal abstract plan solves the determinization of the original problem. Obviously, the PDB heuristic constructed from this pattern is not very informative.

Policy-Based CEGAR

The complications discussed above arise as an artifact of determinization. We now revise the refinement procedure and recover the completeness guarantees from determinization-based CEGAR so that the CEGAR algorithm will eventually find an optimal policy. To this end, we instead inspect optimal abstract policies for the MDP projection with respect to a pattern P. To accommodate this change, the procedure to check for precondition and goal flaws in an abstract policy π_P must now consider all possible executions of the policy in the concrete state space when starting from $s_{\mathcal{I}}$. To properly execute π_P in the concrete state space, we define the concrete partial policy $\pi(s) := \pi_P(s[P])$ if $\pi_P(s[P]) \in \mathcal{A}(s)$ and $\pi(s) = \perp$ otherwise. To check if π has any flaws, we need to verify:

1. Executing π from $s_{\mathcal{I}}$ in the original state space may not lead to a state s in which a precondition flaw with respect to some variable $v \notin P$ occurs: $\perp \neq pre_{\pi(s)}[v] \neq s[v]$.
2. Executing π from $s_{\mathcal{I}}$ cannot lead to a state s for which a goal flaw with respect to some variable $v \notin P$ occurs: $s[P]$ is an abstract goal, but s is not because $s[v] \neq \mathcal{G}[v]$.

The first condition makes sure that π is closed for the initial state and thus constitutes a solution. The second condition makes sure that this solution is also proper.

Theorem 1 *If π_P is an optimal abstract policy and π has no flaws, then π is optimal.*

Proof (sketch). Since precondition flaws do not occur, the abstract state σ is reachable by π_P from $s_{\mathcal{I}}[P]$ only if there is some concrete state s with $s[P] = \sigma$ which is reachable by π from $s_{\mathcal{I}}$. Let f be some function mapping an abstract state σ to such a concrete state $f(\sigma)$, and let $f(s_{\mathcal{I}}[P]) = s_{\mathcal{I}}$.

Considering equation system (1) for the states reachable with π_P from $s_{\mathcal{I}}[P]$ only, it is easy to see that the assignment $x(\sigma) = V^{\pi}(f(\sigma))$ is a solution. As V^{π_P} is the least solution and π_P is optimal, we get $V^*(s_{\mathcal{I}}[P]) = V^{\pi_P}(s_{\mathcal{I}}[P]) \leq V^{\pi}(s_{\mathcal{I}}) \leq V^*(s_{\mathcal{I}})$. Since $V^*(s) \leq V^*(s[P])$ for all states $s \in \mathcal{S}$ (Klößner et al. 2021), π must be optimal. The claim also holds for SSP objectives by using equation system (2) with analogous arguments. $\square$

We instantiate this policy-based CEGAR framework by adapting the pattern generation algorithm by Rovner, Sievers, and Helmert (2019), which operates on multiple disjoint projections. In a nutshell, this algorithm starts with the disjoint pattern collection containing only a singleton pattern for each goal variable. Each pattern is associated with an abstract plan for its corresponding projection. In each iteration, the algorithm collects the flaws for each optimal abstract plan associated with these patterns. A flaw is then chosen uniformly at random and added to the corresponding pattern. To keep the invariant that the patterns are disjoint, if a variable is selected for the refinement that is already present in another pattern, both patterns are merged instead.

We customize their algorithm at three specific points we will talk about in detail in the following. Firstly, we now associate each pattern with an abstract policy for the associated *probabilistic* projection. Secondly, we amend the flaw finding procedure to accept policies instead of plans. Lastly, we consider the extension to *wildcard* policies, which are similar to wildcard plans in their original description.

Computing an optimal abstract policy To compute an abstract policy, we first compute the full pattern database for the projection using topological value iteration (Dai et al. 2011), where the previous PDB is used as an admissible initialization hint to accelerate the convergence. Afterwards, an optimal policy is extracted from the optimal value function by expanding the abstract state space from the initial state. In each step, an optimal action is selected randomly among all greedy operators and its non-goal state successors are marked for expansion. This yields a partial optimal abstract policy for the initial state. For MaxProb, we are forced to expand all greedy operators instead of choosing a single one randomly, as choosing greedy operators arbitrarily may introduce cycles which prevent the policy from being optimal. Afterwards, we explore the generated search graph backwards with duplicate checking, starting from the goal states. When encountering a predecessor, the greedy action responsible for its generation is selected for it by the policy. Because of duplicate checking, the chosen operators do not introduce cycles and the policy is optimal. For pseudocode of this procedure, we refer to Kolobov et al. (2011).

Flaw finding strategies To find the flaws of the optimal abstract policy in the original problem, we search the induced graph of the policy, starting from $s_{\mathcal{I}}$. However, the choice of the search algorithm used to expand this graph is not obvious. Depending on the algorithm used, a flaw might be found much sooner or later during exploration of the policy graph. More importantly, not every flaw is equally likely to be encountered during plan execution. Consider the example of a policy that provokes a flaw with respect to variable v with a probability of 1% and a flaw with respect to variable w with 50% chance. Intuitively, we expect that refining the abstraction with w yields a greater benefit, as the current abstract policy relies on illegal behaviour with a much higher probability. In contrast, refining with v will most likely increase the quality of the abstraction by very little.

The strategy that we deduce from this observation selects a variable with respect to which a flaw happens with highest probability among all candidates. Unfortunately, this strategy is costly in practice, as it involves the computation of n hitting probabilities in the Markov chain induced by the policy, where n is the number of variables that can potentially occur in a flaw. Therefore, we derive three approximations of this strategy:

1. A Monte-Carlo strategy in which successor states are sampled according to their probability (Algorithm 1).
2. A strategy that sorts exploration candidates by increasing likelihood of their generating path (Algorithm 2).
3. Breadth-first search, as a strategy that approximates this behaviour for problems in which the successor distributions are close to uniform.

In the pseudocode, the function GETFLAWS(s) returns all goal and precondition flaws that occur in state s as a set of variables. The function INSERTORUPDATE(Q, S, P) inserts the state s with priority p into the priority queue Q if it is missing, or updates the priority of s with p if this value is higher than the previous priority. We always stop at the first flaw we encounter during the execution.

Wildcard policies Lastly, we briefly consider the extension of policy-based CEGAR to wildcard policies, which are in direct correspondence with wildcard plans. Instead of settling for a single optimal abstract operator, we also collect all abstract operators which are equivalent, i.e. they transition to the same abstract states with the same probability. Exhaustively checking whether one of the policies represented by this wildcard policy is optimal for the original task wastes even more resources than in classical planning, so we avoid this approach. Instead, we employ the greedy algorithm that chooses a single random action in each state that does not lead to a flaw and expands only the successors of this action. If all operators fail, all associated flaws are reported.

Algorithm 1: Monte-Carlo Sampling (MCS) Strategy

```
1:  function FINDFLAWSMCS($\pi, s_0$)
2:      return FINDFLAWSMCREC($\pi, s_0, \emptyset$)
3:
4:  function FINDFLAWSMCSREC($\pi, s, closed$)
5:      if $s \in closed \cup \mathcal{S}_{\mathcal{G}}$ then
6:          return $\emptyset$
7:      $flaws \leftarrow$ GETFLAWS($s$)
8:      if $flaws \neq \emptyset$ then
9:          return $flaws$
10:
11:     $closed = closed \cup \{s\}$
12:     $successors = \{t.\ \mathcal{T}(s, \pi(s), t) > 0\}$
13:     while $successors \neq \emptyset$ do
14:         sample $t \leftarrow successors$ according to $\mathcal{T}$
15:         $successors = successors \setminus \{t\}$
16:         $flaws \leftarrow$ FINDFLAWSMCREC($\pi, t, closed$)
17:         if $flaws \neq \emptyset$ then
18:             return $flaws$
```

Admittedly, we could even go one step further and collect all greedy operators for a state during extraction of the optimal abstract policy, to capture all possible optimal abstract policies. However, we do not pursue this extension for two reasons. First of all, doing so potentially expands a larger part of the abstract state space, as we now have to add all successors of all greedy operators during the policy extraction step. This is required in MaxProb anyway, but causes additional work for the SSP objective and introduces even more overhead in the flaw finding procedure as well. Secondly, it does not suffice to pick greedy operators arbitrarily in MaxProb to obtain an optimal policy, so the greedy algorithm above is not even sound in this setting.

Algorithm 2: Most-Likely Path (MLP) Strategy

```
1:  function FINDFLAWSMLP($\pi, s_0$)
2:      // priority queue, sorted by ascending priority
3:      open $\leftarrow \{\langle s_0, 1\rangle\}$
4:      closed $\leftarrow \emptyset$
5:
6:      while $open \neq \emptyset$ do
7:          choose $\langle s, p\rangle \leftarrow open$ with highest $p$
8:          $open \leftarrow open \setminus \{\langle s, p\rangle\}$
9:          if $s \in closed \cup \mathcal{S}_{\mathcal{G}}$ then
10:             continue
11:         $closed \leftarrow closed \cup \{s\}$
12:         $flaws \leftarrow$ GETFLAWS($s$)
13:         if $flaws \neq \emptyset$ then
14:             return $flaws$
15:
16:         for all $t$ s.t. $\mathcal{T}(s, \pi(s), t) > 0$ do
17:             $p_{new} = p \cdot \mathcal{T}(s, \pi(s), t)$
18:             INSERTORUPDATE($open, t, p_{new}$)
19:
20:     return $\emptyset$
```

Pattern Generation by Hillclimbing

Pattern selection using local search (Haslum et al. 2007) is a conventional approach to generate a pattern collection for PDBs in classical planning. In this framework, a pattern collection is constructed as a local optimum of hillclimbing search in the space of pattern collections. Initially, the search starts with the pattern collection that contains a single variable pattern $\{v\}$ for every goal variable v. The search neighborhood of a pattern C is the set of all possible extensions $C' = C \cup (P \cup \{v\}) \supsetneq C$ of the collection, where $P \in C$ is any pattern of C, augmented by a single new variable $v \notin P$. To rank the neighboring pattern collections, Haslum et al. use a counting approximation which collects a set of sample states from the state space and counts how often the canonical PDB heuristic $h^{\text{can}}_{C'}$ induced by the neighboring pattern collection improves upon the estimate for the sample states compared to the heuristic of the current collection h^{can}_{C}, ranking the pattern collection with the most improvements best. The search terminates with the current collection if the number of improvements for the best candidate is below a user-defined threshold, or if a resource limit is reached.

For SSP problems, we could use this algorithm on the all-outcomes determinization of our task to construct a pattern collection. However, for the search to be meaningful, the heuristic used to rank the pattern collections should be equal or close to the PDB heuristic that we would construct from C as the generated pattern collection, in our case being h^{SSP}_{C} instead of h^{can}_{C}. In general, the classical variant may deviate from the probabilistic variants by an arbitrary amount. Also, a local optimum may be reached very early if the classical variants are used, which again happens in the example of triangle-tireworld depicted in Figure 1, where any pattern collection that includes the location of the vehicle induces an expected cost-to-goal estimate of nc for any state in which the tire is flat in the determinization, where n is the distance to the goal and c is the cost of driving.

Conveniently, the above high-level description solely depends on the definition of the heuristic h^C induced by a pattern collection, so we may generate pattern collections for SSPs using the heuristic h^{SSP}_{C} directly during hillclimbing. However, if we want to use the counting approximation to rank the neighborhood, we need to specify how to sample random states in the problem. Haslum et al. use a random walk from the initial state of specific length for this, which has the significant advantage that only reachable states of the problem are sampled. The length of this random walk is derived from the current cost-to-goal estimate of the initial state in the classical setting, whereas we can use the current expected cost-to-goal estimate in the SSP setting (rounded up), which we found to be sufficient in practice.

While we could also use the heuristic h^{MaxProb}_{C} for hillclimbing in MaxProb, we do not consider this extension for two reasons. Firstly, as already observed by Klößner21, patterns with few variables frequently provide only trivial goal probability estimates of one. This is highly problematic for hillclimbing, which may hence get stuck in a local minimum very early and much easier in MaxProb than in the SSP setting. Secondly, the length of a random walk to collect sample states is difficult to choose in MaxProb, as we have no way to approximate the distance between the initial state and the goal states, possibly leading to strong bias towards either. Due to these difficulties, we only consider hillclimbing for SSP problems in our experiments.

Regarding efficient implementation of this algorithm, Haslum et al. point out that some of the computations involving the PDBs can be done incrementally. In the deterministic case where the canonical PDB heuristic h^{can}_{C} is used, the comparison of $h^C(s)$ with $h^{C'}(s)$ given the sample state s can be implemented by checking whether

$$h^{P'}(s) + \sum_{P \in S \setminus \{P'\}} h^P(s) > h^C(s)$$

for any additive subset $S \in \mathcal{Q}(C')$ that includes P'. Assuming the additive subsets are easy to compute, $h^{P'}(s)$ remains the only unknown in this term. Here, $h^{P'}(s)$ can be computed efficiently by using h^P as a heuristic for A*. We emphasize that the same can be done with any applicable MDP heuristic search algorithm when using h^{SSP}_{C}. Hence, the full PDB does not need to be constructed for all search neighbors. Furthermore, the full PDB for pattern P' can be constructed once necessary by using topological value iteration supplied with h^P as an initialization hint for the value function, leading to faster convergence.

Experiments

We implemented both of our algorithms in our probabilistic planning adaptation of the Fast Downward planner (Helmert 2006), which is publicly available (Klößner et al. 2022). As a baseline for our experiments, we consider systematic pattern generation (Pommerening, Röger, and Helmert 2013) (SYS), as previously used by Klößner et al. (2021) in their experiments with probabilistic pattern databases. In a nutshell, this generator constructs the collection of patterns of size $\leq K$, but prunes some patterns which can be identified as redundant based on the causal graph of the (determinized) problem. We set $K = 2$ in our experiments. We also consider the plan-based variant of the Disjoint CEGAR algorithm with regular (cCG) and wildcard plans (wcCG), and hillclimbing (cHC) on the determinization of the task, using the canonical PDB heuristic to rank pattern collections. For all of these algorithms, we use the base implementations of Fast Downward. Our policy-based CEGAR generator with regular policies (CG) and wildcard policies (wCG) and flaw finding strategies Monte-Carlo Sampling (mcs), Most-Likely Path (mlp) and Breadth-First Search (bfs), as well as our hillclimbing adaptation (HC) were implemented by modifying these implementations where necessary. The generated pattern collection is always used to construct a probabilistic PDB heuristic with additivity or multiplicativity constraints as described by Klößner21. We use the orthogonality criterion for all configurations.

We ran our experiments on a cluster with Intel Xeon E5-2650 v3 processors @2.30 GHz, with a time limit of 1800 seconds and a memory limit of 4GB, using Downward Lab (Seipp et al. 2017). For convergence tests, we used a precision of $\epsilon = 10^{-5}$. For configurations involving randomness, we used 10 different random seeds and averaged the results.

Domains	N	SYS	cCG	CG			wcCG	wCG		
				mcs	mlp	bfs		mcs	mpl	bfs
blocksworld	90	24	25.2	**26**	24.8	25.2	25.6	24	24.8	24
canadian-nomystery	120	76	**90.6**	89	89.2	89.2	76.8	71.6	66.8	66
canadian-rovers	120	103	**111.9**	110.9	110.5	110.3	111.3	110	109.9	110
canadian-tpp	120	58	**74.5**	70.9	73.1	73.2	67.9	72.1	73.2	72.1
coresec	30	**16**	12	14.2	12	13.2	12	14	12	13.4
drive	90	**90**	**90**	**90**	**90**	**90**	**90**	89.90	**90**	**90**
elevators	90	76	85.1	84.7	84.8	84.9	**86.8**	72.3	72.9	72.8
exploding-blocksworld	84	**42**	40	40	40.1	40.1	39.9	38.6	38.9	38.9
random	72	43	44.6	44.9	**45**	45	38.6	38.5	38.2	38.6
rectangle-tireworld	36	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**	**12**
schedule	60	30	35.4	35.9	**36**	**36**	34.5	35.1	35.1	35.4
search-and-rescue	90	65	78	78.1	78	78	78	**78.2**	78	78
triangle-tireworld	120	57	59	**62**	**62**	**62**	52.8	**62**	**62**	**62**
zenotravel	42	**18**	**18**	**18**	**18**	**18**	**18**	17	16.7	16.6
canadian-nomystery	120	15	47.8	47.7	47.1	47.2	14.9	49.4	51	**51.3**
canadian-rovers	120	101	**110.3**	108.8	108.7	108.8	101	109	108	108.5
canadian-tpp	120	72	**92.7**	92.1	92.1	91.9	71.4	87.1	89.8	89.8
drive	15	**15**	**15**	**15**	**15**	**15**	**15**	13	13	13
exploding-blocksworld	15	**8**	4.1	4.2	4.6	4.4	4.4	4.2	4.7	4.7
search-and-rescue	15	5	5.5	**5.9**	5	5	5.4	**5.9**	5	5.1
Sum	1569	926	**1052.9**	1051.9	1049.8	1051.6	956.3	1005.1	1005.0	1003.9

Table 1: Coverage (sum) for the acyclic (top rows), cyclic (bottom rows) MaxProb benchmarks. Coverage is averaged over ten random seeds for the CEGAR configurations. The highest coverage in a domain is highlighted in boldface. Domains with full or zero coverage for all configurations are omitted to decrease table size. N is the total number of problems per domain.

For configurations requiring an LP solver, we used CPLEX 12.6.3.0. We limit the time for pattern generation to a maximum of 180 seconds for all of these algorithms. The hill-climbing configurations collect 1000 sample states to compute the counting approximation. For all pattern generators, we fixed the maximum size of a single PDB to 1 million states, and the total collection size to 10 million states.

Benchmarks

We use the set of benchmarks used by Klößner21 for their evaluation in the MaxProb and SSP settings, briefly described in the following. The benchmark set includes the PPDDL domains from the IPPC 2004 & 2008. For domains present in both iterations, we used only the newer version. We consider those domains with avoidable dead-ends for the SSP setting and domains with unavoidable dead-ends for the MaxProb setting. We also experimented with a finite-penalty approach for domains with unavoidable dead-ends in the SSP setting. However, the results add nothing new in comparison, so we omit them due to space constraints. For all SSP domains, we assume unit costs.

For MaxProb, the benchmark also contains domains from resource-constrained planning and an automated penetration testing domain (coresec). Further, the MaxProb benchmark contains *acyclic* variants of each domain except coresec, which is naturally acyclic. These are obtained by introducing a finite budget which is consumed by all actions via a variable at PPDDL level. These include variants of IPPC domains which naturally only contain avoidable dead-ends, since unavoidable dead-ends are introduced in the process.

For acyclic problems, we use AO* as the heuristic search algorithm, whereas we used iLAO* (Hansen and Zilberstein 2001) with FRET (Kolobov et al. 2011) for cyclic problems. All algorithms are run until ϵ-convergence, i.e. the values of states reachable by the current greedy policy did not change by more than ϵ compared to the previous value update.

MaxProb Analysis

We first consider MaxProb analysis. The coverage table for these experiments is depicted in Table 1. Firstly, we see that the CEGAR configurations achieve a substantially higher coverage than the previous state of the art in systematic pattern generation, which beats CEGAR only in two domains. Among the different CEGAR configurations, the wildcard variations consistently perform worse than their regular counterpart. For wildcard plans, this does not seem too surprising, as here it is even more likely for classical CEGAR to terminate early with a working optimal plan for the determinization, which only amplifies the previously discussed issues. For the variations computing wildcard policies, we notice a nearly fivefold increase in average construction time for the configurations using the sampling strategy to find flaws, and similarly for the other strategies. This clearly indicates that an enormous overhead is introduced when considering wildcard policies during the flaw finding procedure, which is rarely compensated for. All in all, these results discourage the usage of wildcard plans, as well as wildcard policies, in MaxProb.

The total coverage of the regular CEGAR configurations is nearly identical. If we go into more detail and compare the average number of states evaluated during the search between our best policy-based CEGAR configuration and plan-based CEGAR, we can see that policy-based CEGAR has an extreme advantage over plan-based CEGAR in par-

Domains	N	h^{roc}	SYS	cHC	HC	cCG	CG-mcs	CG-mlp	CG-bfs
blocksw	8	**154,108.0**	163,292.0	161,748.0	160,888.3	9,180,857.1	10,698,053.3	12,001,092.1	11,844,380.7
elevators	14	130,611.9	116,976.7	115,784.9	114,372.6	104,147.5	101,571.3	100,976.1	**100,953.3**
random	5	14,354.2	17,695.0	2,307.6	**740.8**	2,308,825.0	4,712,126.4	4,712,126.4	4,712,126.4
schedule	3	**60.7**	**60.7**	**60.7**	**60.7**	**60.7**	**60.7**	**60.7**	**60.7**
tri-tirew	5	168,412.8	221,642.4	92,423.4	**52,055.0**	221,646.1	69,441.0	**52,055.0**	81,601.7
zenotrvl	3	18,944.7	9,972.7	**97.7**	100.0	617.1	98.9	98.8	98.8
Mean	38	81,082.0	88,273.3	62,070.4	**54,702.9**	1,969,359.0	2,596,892.0	2,811,068.2	2,789,870.3

Table 2: Evaluated states for the SSP benchmarks. Each entry reports the average over domain instances covered by all algorithms, as well as over ten random seeds (hillclimbing, CEGAR). The lowest number per domain is highlighted in boldface. N is the number of commonly covered instances per domain.

Domains	N	h^{roc}	SYS	cHC	HC	cCG	CG		
							mcs	mlp	bfs
blocksw	15	**12**	8	**12**	**12**	8	8	8	8
elevators	15	14	14	14	14	14.8	**14.9**	14.8	14.8
random	15	**8**	7	**8**	7.1	4.7	6	7	6.7
schedule	15	3	**4**	**4**	**4**	**4**	**4**	**4**	**4**
tri-tirew	10	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
zenotrvl	15	3	3	3	**4**	3.3	3.3	3.2	3.2
Sum	85	45	41	46	**46.1**	39.8	41.2	42	41.7

Table 3: Coverage (sum) for the SSP benchmarks. Coverage is averaged over ten random seeds for hillclimbing and CEGAR configurations. The highest coverage in a domain is highlighted in boldface. Wildcard CEGAR configurations perform considerably worse and are omitted from this table. N is the total number of problem instances for each domain.

ticular domains, including triangle-tireworld (-97.5%), as well as in the automated penetration testing domain coresec (-54.8%), for which plan-based CEGAR also runs into the problem of pathological early termination. We conclude that policy-based CEGAR is a good alternative choice over plan-based CEGAR in domains where such early termination is a common occurrence. However, we also observe an increase for some domains, for example canadian-rovers ($+20.8\%$ acyclic, $+28.3\%$ cyclic), where this problem rarely occurs.

Regarding flaw finding procedures, we find only few domains where we can see a clear distinction. Our sampling strategy performs best, yet the advantage over our other strategies seems marginal in total. The difference in the average number of evaluated states in all problems measures below 2% between each of the strategies. We therefore conclude that the flaw finding strategy only has a negligible influence on the quality of the constructed pattern collection.

SSP Problems

Finally, we present our results for the SSP setting, which includes the hillclimbing algorithms. Here, we also compare against the regrouped operator-counting heuristic h^{roc} (Trevizan, Thiébaux, and Haslum 2017), a state-of-the-art heuristic for this setting which is competitive with probabilistic PDBs. The coverage table is depicted in Table 3. Pattern construction by hillclimbing produces the best results in this setting, also outperforming h^{roc}, which achieves slightly better coverage than the simplistic systematic pattern generation. The classical hillclimbing variant achieves virtually the same coverage as our adaptation, which however evaluates fewer states on average in the benchmarks random and triangle-tireworld, as can be seen in Table 2.

Regarding pattern generation by CEGAR, the variations which use wildcard plans and policies again perform much worse due to the introduced overhead. On the other hand, policy-based CEGAR achieves a slightly higher coverage than plan-based CEGAR here, due to covering more instances in the problem domain random. For triangle-tireworld, policy-based CEGAR again evaluates much fewer states on average (-76.5%), yet this advantage does unfortunately not lead to a higher coverage. All in all, we again make the observation that policy-based CEGAR leads to noticeable improvements in a domain where plan-based CEGAR terminates early, but behaves similarly otherwise.

The hillclimbing configurations tend to perform even better than CEGAR in this setting. However, we point out that in classical planning, Rovner, Sievers, and Helmert (2019) achieved significantly better results when combining the PDBs generated by the disjoint CEGAR algorithm with saturated cost partitioning (Seipp and Helmert 2014; Seipp, Keller, and Helmert 2020), as the patterns produced by CEGAR are typically too large to be combined using ordinary additivity constraints. As of yet, it is unclear whether or how we can pursue a similar strategy to combine the patterns produced by CEGAR, so it is not yet safe to say that hillclimbing is all in all the better strategy to use for SSP problems. For now, our results suggest this is the case.

Conclusion

We introduced two novel pattern generation algorithms designed for the probabilistic planning settings of MaxProb and stochastic shortest-path problems. Both algorithms improve considerably over systematic pattern generation, the previous state-of-the art in terms of pattern generation algorithms for these settings. Our policy-based CEGAR algorithm recovers the convergence guarantees of plan-based CEGAR, and our experiments indicate a significant improvement in problem domains where this convergence poses an issue. Our adaption of hillclimbing improves upon the determinization-based approach and constitutes a simple yet effective pattern construction technique for the SSP setting which performs best in our experiments.

Acknowledgements

This work was funded by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science).

References

Bertsekas, D. 1995. *Dynamic Programming and Optimal Control, (2 Volumes)*. Athena Scientific.

Bonet, B.; and Geffner, H. 2003. Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In *Proc. ICAPS'03*, 12–21.

Bryce, D.; Kambhampati, S.; and Smith, D. E. 2006. Sequential Monte Carlo in probabilistic planning reachability heuristics. In *Proc. ICAPS'06*, 233–242.

Clarke, E.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5): 752–794.

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

Dai, P.; Mausam; Weld, D. S.; and Goldsmith, J. 2011. Topological Value Iteration Algorithms. *JAIR*, 42: 181–209.

Domshlak, C.; and Hoffmann, J. 2007. Probabilistic Planning via Heuristic Forward Search and Weighted Model Counting. *JAIR*, 30: 565–620.

E-Martín, Y.; Rodríguez-Moreno, M. D.; and Smith, D. E. 2014. Progressive heuristic search for probabilistic planning based on interaction estimates. *Expert Systems - The Journal of Knowledge Engineering*, 31(5): 421–436.

Edelkamp, S. 2001. Planning with Pattern Databases. In *Proc. ECP'01*, 13–24.

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

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *Proc. AAAI'07*, 1007–1012.

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

Hermanns, H.; Wachter, B.; and Zhang, L. 2008. Probabilistic CEGAR. In Gupta, A.; and Malik, S., eds., *Proc. the 20th International Conference on Computer Aided Verification (CAV 2008)*, volume 5123 of *Lecture Notes in Computer Science*, 162–175. Springer-Verlag.

Holte, R. C.; Felner, A.; Newton, J.; Meshulam, R.; and Furcy, D. 2006. Maximizing over multiple pattern databases speeds up heuristic search. *AI*, 170(16-17): 1123–1136.

Keyder, E.; and Geffner, H. 2008. The HMDP Planner for Planning with Probabilities. In *IPC 2008 planner abstracts*.

Klauck, M.; Steinmetz, M.; Hoffmann, J.; and Hermanns, H. 2020. Bridging the Gap Between Probabilistic Model Checking and Probabilistic Planning: Survey, Compilations, and Empirical Comparison. *JAIR*, 68: 247–310.

Klößner, T.; and Hoffmann, J. 2021. Pattern Databases for Stochastic Shortest Path Problems. In *Proc. the 14th Annual Symposium on Combinatorial Search (SOCS'21)*, 131–135. AAAI Press.

Klößner, T.; Steinmetz, M.; Torralba, Á.; and Hoffmann, J. 2022. Code and Benchmarks of the ICAPS'22 submission "Pattern Selection Strategies for Pattern Databases in Probabilistic Planning". https://doi.org/10.5281/zenodo.6382098.

Klößner, T.; Torralba, Á.; Steinmetz, M.; and Hoffmann, J. 2021. Pattern Databases for Goal-Probability Maximization in Probabilistic Planning. In *Proc. ICAPS'21*, 80–89.

Kolobov, A.; Mausam; Weld, D. S.; and Geffner, H. 2011. Heuristic Search for Generalized Stochastic Shortest Path MDPs. In *Proc. ICAPS'11*.

Korf, R. E. 1997. Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. In Kuipers, B. J.; and Webber, B., eds., *Proc. the 14th National Conference of the American Association for Artificial Intelligence (AAAI'97)*, 700–705. Portland, OR: MIT Press.

Little, I.; and Thiébaux, S. 2006. Concurrent Probabilistic Planning in the Graphplan Framework. In *Proc. ICAPS'06*, 263–273.

Pommerening, F.; Röger, G.; and Helmert, M. 2013. Getting the Most Out of Pattern Databases for Classical Planning. In *Proc. IJCAI'13*.

Puterman, M. L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley. ISBN 978-0-47161977-2.

Rovner, A.; Sievers, S.; and Helmert, M. 2019. Counterexample-Guided Abstraction Refinement for Pattern Selection in Optimal Classical Planning. In *Proc. ICAPS'19*, 362–367.

Seipp, J. 2012. *Counterexample-guided Abstraction Refinement for Classical Planning*. Master's thesis, University of Freiburg, Germany.

Seipp, J.; and Helmert, M. 2014. Diverse and Additive Cartesian Abstraction Heuristics. In *Proc. ICAPS'14*.

Seipp, J.; and Helmert, M. 2018. Counterexample-Guided Cartesian Abstraction Refinement for Classical Planning. *JAIR*, 62: 535–577.

Seipp, J.; Keller, T.; and Helmert, M. 2020. Saturated Cost Partitioning for Optimal Classical Planning. *JAIR*, 67: 129–167.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Steinmetz, M.; Hoffmann, J.; and Buffet, O. 2016a. Goal Probability Analysis in MDP Probabilistic Planning: Exploring and Enhancing the State of the Art. *JAIR*, 57: 229–271.

Steinmetz, M.; Hoffmann, J.; and Buffet, O. 2016b. Revisiting Goal Probability Analysis in Probabilistic Planning. In *Proc. ICAPS'16*.

Trevizan, F. W.; Teichteil-Königsbuch, F.; and Thiébaux, S. 2017. Efficient solutions for Stochastic Shortest Path Problems with Dead Ends. In Elidan, G.; Kersting, K.; and Ihler, A. T., eds., *Proc. the 33rd Conference on Uncertainty in Artificial Intelligence (UAI'17)*. AUAI Press.

Trevizan, F. W.; Thiébaux, S.; and Haslum, P. 2017. Occupation Measure Heuristics for Probabilistic Planning. In *Proc. ICAPS'17*, 306–315.

Trevizan, F. W.; Thiébaux, S.; Santana, P. H.; and Williams, B. 2017. I-dual: Solving Constrained SSPs via Heuristic Search in the Dual Space. In *Proc. IJCAI'17*, 4954–4958.

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