Learning Generalized Policy Automata for Relational Stochastic Shortest Path Problems

Rushang Karia, Rashmeet Kaur Nayyar, and **Siddharth Srivastava**
School of Computing and Augmented Intelligence, Arizona State University, U.S.A.
{Rushang.Karia,rmnayyar,siddharths}@asu.edu

Abstract

Several goal-oriented problems in the real-world can be naturally expressed as Stochastic Shortest Path Problems (SSPs). However, the computational complexity of solving SSPs makes finding solutions to even moderately sized problems intractable. Currently, existing state-of-the-art planners and heuristics often fail to exploit knowledge learned from solving other instances. This paper presents an approach for learning *Generalized Policy Automata* (GPA): non-deterministic partial policies that can be used to catalyze the solution process. GPAs are learned using relational, feature-based abstractions, which makes them applicable on broad classes of related problems with different object names and quantities. Theoretical analysis of this approach shows that it guarantees completeness and hierarchical optimality. Our experiments show that this approach effectively learns broadly applicable policy knowledge in a few-shot fashion and significantly outperforms state-of-the-art SSP solvers on test problems whose object counts are far greater than those used during training.

1 Introduction

Goal-oriented Markov Decision Processes (MDPs) expressed as Stochastic Shortest Path problems (SSPs) have been the subject of active research since they provide a convenient framework for modeling the uncertainty in action execution that often arises in the real-world. Recently, research in deep learning has demonstrated success in finding solutions to goal-oriented MDPs using image-based state representations [Tamar *et al.*, 2016; Pong *et al.*, 2018; Levy *et al.*, 2019]. However, these methods require significant human-engineering effort in finding transformations like grayscale conversion, etc., to yield representations that facilitate learning. A large number of practical problems however, are more intuitively expressed using relational representations and have been widely studied in the literature. Consider a planetary rover whose mission is to collect all rocks of interest from a planet's surface and deliver them to the base for analysis. Such a problem objective is not easily described in an image-based representation (e.g., visibility is affected by line of sight) but lends itself to a relational description. Converting these existing problems to suitable image-based representations would be counter-productive and difficult. Furthermore, they either require large amounts of training data and/or are unable to provide guarantees of completeness and/or convergence.

Many real-world problems can be readily expressed as SSPs and polynomial-time algorithms are available for solving them. A common theme among different SSP solvers is to use a combination of pruning strategies (e.g., heuristics [Hansen and Zilberstein, 2001]) that can eliminate large parts of the search space from consideration, thereby reducing the computational effort expended. Despite such optimizations, a major hurdle that SSP solvers face is the "curse-of-dimensionality" since the state spaces grow exponentially as the total number of objects increases. The pruning strategies employed by these SSP solvers fail because they are "stateless" in the sense that they do not store, and consequently fail to re-use, knowledge that could have been exploited by solving similar problems.

For rover problems, the total number of possible states grows exponentially as a factor of the total number of locations and rocks. Existing SSP solvers would have difficulty scaling to problems with many locations and/or rocks. A better way to tackle this problem of state space explosion would be to compute a simple *generalized policy*: move the rover to the closest available location with a rock, try loading the rock until it succeeds, navigate back to the base, unload it, and re-iterate this process until all the rocks are at the base.

Such a generalized policy can be used to solve many different SSP problems with large numbers of objects that share similar objectives. Furthermore, such policies appear to be easily computable using solutions of problems with few numbers of objects. Recent work has demonstrated that it is possible to learn such generalized policies. Toyer *et al.*; Groshev *et al.*; Garg *et al.* [2018; 2018b; 2020] use deep learning to learn generalized reactive policies. Bonet *et al.* [2009] automatically derive controllers that represent generalized reactive policies. A key limitation of these approaches is the lack of any theoretical guarantees of finding a solution or its optimality even if one is guaranteed to exist. In this paper, we show that such policies can be learned with guarantees of completeness and hierarchical optimality using solutions of very few, *small* problems with few objects.

The primary contribution of this paper is a novel approach for few-shot learning *Generalized Policy Automata* (GPAs) using solutions of SSP instances with small object counts. GPAs are non-deterministic partial policies that represent generalized knowledge that can be applied to problems with different object names and larger object counts. This process uses logical feature-based abstractions to lift instance-specific information like object names and counts while preserving the relationships between objects in a way that can be used to express generalized knowledge. GPAs learned using our approach can be used to *accelerate any model-based SSP solver*. Our approach uses GPAs to prune out large sets of actions in different, related but larger SSPs. We prove that our approach is complete and guarantees hierarchical optimality. Empirical analysis on a range of well-known benchmark domains used in various planning competitions and robotics shows that our approach few-shot learns GPAs and convincingly outperforms existing state-of-the-art (SOA) SSP solvers and does so without compromising the quality of the solutions found.

The rest of this paper is organized as follows: The next section provides the necessary background. Sec. 3 describes our approach for using example policies in conjunction with abstractions to learn GPAs and use them for solving SSPs. We present our experimental setup and discuss obtained results in Sec. 4. Sec. 5 provides a description of related work in the area. Finally, Sec. 6 states the conclusions that we draw upon from this work followed by a brief description of future work.

2 Background

Our problem setting considers SSPs expressed in a symbolic description language such as the Probabilistic Planning Domain Definition Language (PPDDL) [Younes *et al.*, 2005]. Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{A} \rangle$ be a problem *domain* where $\mathcal{P}$ is a set of predicates and $\mathcal{A}$ is a set of parameterized actions. Object *types*, such as those used in PPDDL, can be equivalently represented using unary predicates. A relational SSP problem for a domain $\mathcal{D}$ is then defined as a tuple $P = \langle O, S, A, s_0, g, T, C \rangle$ where O is a set of objects. A fact is the instantiation of a predicate $p \in \mathcal{P}$ with the appropriate number of objects from O. A state s is a set of true facts and the state space S is defined as all possible sets of true facts derived using $\mathcal{D}$ and O. Similarly, the action space A is instantiated using $\mathcal{A}$ and O. $T : S \times A \times S' \rightarrow [0, 1]$ is the transition function and $C : S \times A \times S \rightarrow \mathbb{R}^+$ is the cost function. An entry $t(s, a, s') \in T$ defines the probability of executing action a in a state s and ending up in a state s' where $a \in A$, $s, s' \in S$, and $c(s, a, s') \in C$ indicates the cost incurred while doing so. Naturally, $\sum_{s'} t(s, a, s') = 1$ for any $s \in S$ and $a \in A$. Note that a refers to the instantiated action $a(o_1, \ldots, o_n)$, where $o_1, \ldots, o_n \in O$ are the action *parameters*. We omit the parameters when it is clear from context. $s_0 \in S$ is a *known* initial state and g is the goal condition expressed as a first-order logic formula. A goal state s_g is a state s.t. $s_g \models g$. $c(s_g, a, s_g) = 0$ and $t(s_g, a, s_g) = 1$ for all such goal states for any action a. Additionally, termination (reaching a state s.t. $s \models g$) in an SSP is inevitable making the length of the horizon unknown but finite [Bertsekas and Tsitsiklis, 1996].

Running example: The planetary rover example discussed earlier can be expressed using a domain that consists of parameterized predicates *connected*(l_x, l_y), *in-rover*(r_x), *rock-at*(r_x, l_x), and actions load(r_x, l_x), unload(r_x, l_x), and move(l_x, l_y). Object types can be denoted using unary predicates *location*(l_x) and *rock*(r_x). l_x, l_y, and r_x are *parameters* that can be instantiated with different locations and rocks, allowing an easy way to express many different problems with different object names and counts. Actions dynamics are described using closed-form probability distributions (e.g. loading a rock could be modeled to fail with a probability of 0.2) and this forms the transition function. A simplified SSP problem that ignores connectivity and consists of two locations, a base location, and two rocks can be described using a set of objects $O = \{l_1, l_2, \mathit{base}, r_1, r_2\}$. A state in this SSP s_{eg} that describes the situation where r_2 is being carried by the rover and r_1 is at l_2 can be written as $s_{eg} = \{\mathit{location}(l_1), \mathit{location}(l_2), \mathit{location}(\mathit{base}), \mathit{rock}(r_1), \mathit{rock}(r_2), \mathit{in\text{-}rover}(r_2), \mathit{rock\text{-}at}(r_1, l_2)\}$. The goal of delivering all the rocks to the base can be expressed as $\forall r_x\ \mathit{rock}(r_x) \wedge \mathit{rock\text{-}at}(r_x, \mathit{base})$. Actions are assumed to have a unit-cost when executed in a non-goal state and 0 otherwise.

A solution to an SSP is a deterministic policy $\pi : S \rightarrow A$ which is a mapping from states to actions. A *complete proper policy* is one for which the goal is guaranteed to be reachable from all possible states. By definition, SSPs must have at least one *complete* proper policy [Bertsekas and Tsitsiklis, 1996]. This can be overly limiting in practice since such a formulation does not model dead-end states. A weaker formulation of an SSP stipulates that the goal must be reachable with a probability of 1 from s_0 i.e. whose solution is a *partial* proper policy from s_0 that is defined for every reachable state from s_0. To use such a formulation, we focus on a broader class of relaxed SSPs called Generalized SSPs [Kolobov *et al.*, 2012] that allow the presence of dead-end states (whose costs are infinite) and only require the existence of at least one *partial* proper policy from s_0. Henceforth, we use the term SSPs to refer to Generalized SSPs and focus only on *partial* proper policies.

The value of a state s when using a policy π is the expected cost when starting in s and following π thereafter [Sutton and Barto, 1998]:

$$v_\pi(s) = \sum_{s' \in S} t(s, \pi(s), s')[c(s, \pi(s), s') + v_\pi(s')]$$

This equation is the *Bellman* equation for the value of a state under the policy π. The *optimal* policy π_* is one s.t. $v_{\pi_*}(s) \leq v_\pi(s)$ for any policy π. The *Bellman optimality* equation (or the optimal state value function v_*) can be similarly expressed as [Sutton and Barto, 1998]:

$$v_*(s) = \min_{a \in A} \sum_{s' \in S} t(s, a, s')[c(s, a, s') + v_*(s')]$$

Once v_* has been computed, the optimal policy can be extracted using $\pi_*(s) = \arg\min_a v_*(s)$. Thus, given an SSP, the objective is to compute a policy that reaches the goal state from the initial state while minimizing the expected cost of doing so. SSP solvers often compute partial proper policies $\pi_*(s_0)$ by iteratively improving state-value estimates using the Bellman equations. Naturally, for any improper policy π, $v_\pi(s_0) = \infty$. SSP solvers have, under certain conditions, been proved to converge to a policy that is ϵ-consistent with the optimal policy [Hansen and Zilberstein, 2001; Bonet and Geffner, 2003].

A policy π for an SSP P can be represented as a concrete directed hypergraph $G_\pi = \langle V, E\rangle$ where $V = S$ is the vertex set comprising states of P and $E \subseteq V \times \mathcal{P}(V) \times A$ is the set of directed hyperedges s.t. each hyperedge $e \in E$ is a tuple $(e_{\textit{start}}, e_{\textit{results}}, e_{\textit{action}})$ representing the start vertex, a set of result vertices, and an action label. G_π can easily be constructed using the transition function T of P by adding vertices and hyperedges corresponding to each transition from T starting from $\pi(s_0)$. It is easy to see that such a hypergraph representation of π is a non-deterministic finite-state automaton whose states are bounded by the total number of vertices in G_π.

Let f be a feature and F be a set of features. We use feature-based abstractions to lift problem-specific characteristics like object names and numbers in order to facilitate the learning of generalized knowledge that can be applied to problems irrespective of differences in such characteristics. We define *state abstraction* as a function $\alpha : F, S \rightarrow \overline{S}$ that transforms the concrete state space S for an SSP into an abstract state space $\overline{S}$ using the feature set F. Similarly, *action abstraction* $\beta : F, S, A \rightarrow \overline{A}$ transforms the action space to an abstract action space using the feature set F. In this paper, we utilize canonical abstraction [Sagiv *et al.*, 2002] to compute such feature-based representations of concrete states and actions. We define feature sets in more detail in Sec. 3.2.

3 Our Approach

Our objective is to exploit knowledge from solutions of SSP instances with small object counts to learn Generalized Policy Automata (GPAs) that allow effective pruning of the search space of related SSPs with larger object counts. We accomplish this by using solutions to a small set of training instances that are easily solvable using existing SSP solvers, and using feature-based canonical abstractions to learn a GPA that encodes generalized partial policies and serves as a guide to prune the set of policies under consideration. We define GPAs and provide a brief description of canonical abstractions in Sec. 3.1 and Sec. 3.2 respectively, and describe our process to learn a GPA in Sec. 3.3. We then describe our method (Alg. 1) for solving SSPs in Sec. 3.4.

3.1 Generalized Policy Automata

We introduce Generalized Policy Automata (GPAs), which are compact and expressive non-deterministic finite-state automata that encode generalized knowledge and can be represented as directed hypergraphs. GPAs impose hierarchical constraints on the state space of an SSP and prune the action space under consideration, thus reducing the computational effort of solving larger related SSP instances. We now formally define GPAs below.

Definition 3.1 (Abstract states and abstract actions)**.** Given a concrete directed hypergraph $G_\pi = \langle V, E\rangle$ that represents a policy π for an SSP P, abstraction functions α and β, and feature sets F_α and F_β, a set of abstract states $\overline{S}$ is defined as $\{\alpha(F_\alpha, s) | s \in V\}$. Similarly, a set of abstract actions $\overline{A}$ is defined as $\{\beta(F_\beta, s, a) | s \in V, e \in E, s \in e_{\textit{start}}, a \in e_{\textit{action}}\}$.

Definition 3.2 (Generalized Policy Automaton)**.** Given sets of abstract states $\overline{S}$ and abstract actions $\overline{A}$, a Generalized Policy Automaton (GPA) $\overline{G} = \langle \overline{V}, \overline{E}\rangle$ is a non-deterministic finite-state automaton that can be represented as an abstract directed hypergraph where $\overline{V} = \overline{S}$ is the vertex set and $\overline{E} \subseteq \overline{V} \times \mathcal{P}(\overline{V}) \times \overline{A}$ is the set of directed hyperedges s.t. each hyperedge $\overline{e} \in \overline{E}$ is a tuple $(\overline{e}_{\textit{start}}, \overline{e}_{\textit{results}}, \overline{e}_{\textit{action}})$ representing a start vertex, a set of result vertices, and an action label.

3.2 Canonical Abstraction

A key ingredient in learning generalized knowledge using feature-based abstractions is using *rich* feature sets that facilitate the learning of compact generalized policies. We use canonical abstractions for automatically synthesizing a set of such rich, domain-independent features.

Canonical abstractions, commonly used in program analysis, have been shown to be useful in generalized planning [Srivastava *et al.*, 2011; Karia and Srivastava, 2021]. Given a predefined set of abstraction predicates, canonical abstractions group together different objects based on the subset of abstraction predicates that they satisfy in a state. Each subset of abstraction predicates is known as a *summary* element. We used the set of all unary predicates as abstraction predicates in this paper.[1] We use these abstraction predicates to obtain a set of features as described later in the paper.

Let ψ be a summary element, then, we define $\phi_\psi(s)$ as a function that returns the set of objects that satisfy ψ in a concrete state s. An object o belongs to exactly one summary element in any given concrete state s and this is the *maximal* summary element for that object. Similarly, for any given predicate $p_n \in \mathcal{P}$ where n is the arity, $\phi_{p_n}(\psi_1, \ldots, \psi_n)$ is defined as the set of all n-ary predicates in s that are consistent with the summary elements composing the predicate $p_n(\psi_1, \ldots, \psi_n)$, i.e., $\phi_{p_n(\psi_1,\ldots,\psi_n)}(s) = \{p_n(o_1, \ldots, o_n) | p_n(o_1, \ldots, o_n) \in s, o_i \in \phi_{\psi_i(s)}\}$.

The value of a summary element ψ in a concrete state s is given as $\max(2, |\phi_\psi(s)|)$ to indicate whether there are 0, 1, or greater than 1 objects satisfying the summary element. Since relations between objects become imprecise when grouped as summary elements, the value of a predicate $p_n(\psi_1, \ldots, \psi_n)$ in s is determined using three-valued logic and is represented as 0 if $\phi_{p_n(\psi_1,\ldots,\psi_n)}(s) = \{\}$, as 1 if $|\phi_{p_n(\psi_1,\ldots,\psi_n)}(s)| = |\phi_{\psi_1}(s) \times \ldots \times \phi_{\psi_n}(s)|$, and $\frac{1}{2}$ otherwise.

Let Ψ be the set of all summary elements and $\mathcal{P}_i$ be the set of all predicates $p \in \mathcal{P}$ of arity i for a domain D, then $\overline{\mathcal{P}}_i = \mathcal{P}_i \times [\Psi]^i$ is the set of all possible relations of arity i between summary elements. We define the feature set for state abstraction as $F_\alpha = \Psi \cup_{i=2}^{N} \overline{\mathcal{P}}_i$ where N is the maximum arity of any predicate in D. We define state abstraction

[1] 0-ary predicates are represented as unary predicates with a default "phantom" object.

Algorithm 1 GPA-accelerated SSP Solver

```
Require: SSP $P = \langle O, S, A, s_0, g, T, C\rangle$, GPA $\overline{G} = \langle \overline{V}, \overline{E}\rangle$, Feature Sets $F_\alpha, F_\beta$
 1: $C' = C$ {copy over the cost function of $P$}
 2: for $(s, a, s') \in S \times A \times S$ do
 3:     $\overline{s}, \overline{a}, \overline{s'} \leftarrow \alpha(F_\alpha, s), \beta(F_\beta, s, a), \alpha(F_\alpha, s')$
 4:     if there is no edge $\overline{e} \in \overline{E}$ s.t. $\overline{e}_{start} = \overline{s}, \overline{s'} \in \overline{e}_{results}$, and $\overline{e}_{action} = \overline{a}$ then
 5:         $C'[s, a, s'] = \infty$
 6:     end if
 7: end for
 8: $P|_{\overline{G}} = \langle O, S, A, s_0, g, T, C'\rangle$
 9: $v_{P|_{\overline{G}}}, \pi_{P|_{\overline{G}}} \leftarrow$ SSP solver($P|_{\overline{G}}$)
10: if $\pi_{P|_{\overline{G}}}$ is a partial proper policy then
11:     return $\pi_{P|_{\overline{G}}}$
12: else
13:     Initialize $v_P$ using $v_{P|_{\overline{G}}}$
14:     $v_P, \pi_P \leftarrow$ SSP solver($P$)
15:     return $\pi_P$
16: end if
```

$\alpha(F_\alpha, s)$ for a given concrete state s to return an abstract state $\overline{s}$ as a total valuation of F_α using the process described above. Similarly, we define the feature set for action abstraction as $F_\beta = \Psi$. The action abstraction $\beta(F_\beta, s, a)$ for a concrete action $a(o_1, \ldots, o_n)$ when applied to s returns an abstract action $\overline{a}(\psi_1, \ldots, \psi_n)$ where $\overline{a} \equiv a$ and ψ_i is the summary element that object o_i satisfies, i.e., $o_i \in \phi_{\psi_i}(s)$ for $\psi_i \in \Psi$.

Example Consider the state s_{eg} from the running example (Sec. 2). Let the summary elements computed using canonical abstraction be $\psi_1 = \{\textit{location}\}$, $\psi_2 = \{\textit{rock}\}$, and $\psi_3 = \{\textit{in-rover}, \textit{rock}\}$ and let $\Psi_{eg} = \{\psi_1, \psi_2, \psi_3\}$ be the set of all summary elements. We can compute $\phi_{\psi_1}(s_{eg}) = \{\textit{base}, l_1, l_2\}$, $\phi_{\psi_2}(s_{eg}) = \{r_1\}$, and $\phi_{\psi_3}(s_{eg}) = \{r_2\}$. Similarly, $\phi_{\textit{rock-at}(\psi_1,\psi_1)} = \{\}$, $\phi_{\textit{rock-at}(\psi_1,\psi_2)} = \{\}$, $\phi_{\textit{rock-at}(\psi_2,\psi_1)} = \{\textit{rock-at}(r_1, l_2)\}$, and so on. The abstract state $\overline{s}_{eg} = \{\psi_1 = 2, \psi_2 = 1, \psi_3 = 1, \textit{rock-at}(\psi_1, \psi_1) = 0, \textit{rock-at}(\psi_1, \psi_2) = 0, \textit{rock-at}(\psi_2, \psi_1) = \frac{1}{2}, \ldots\}$. Similarly, for an action, unload$(r_2, \textit{base})$, in s_{eg}, the abstract action would be unload(ψ_3, ψ_1).

3.3 Learning GPAs

It is well-known that solutions to small problems can be used to construct generalized control structures that can assist in solving larger problems. We adopt a similar strategy of the learn-from-small-examples approach [Wu and Givan, 2007; Karia and Srivastava, 2021] and compute GPAs iteratively from a small training set containing solutions of similar SSP instances as outlined below.

To form our training set, we use a library of solution policies $\Pi = \{\pi_1, \ldots, \pi_n\}$ for *small* problems $P_1, \ldots, P_n$ that can be easily computed by existing SOA SSP solvers. We represent each policy $\pi_i \in \Pi$ as a concrete directed hypergraph and construct a training set $\mathcal{T} = \{G_{\pi_1}, \ldots, G_{\pi_n}\}$. We initialize an initial empty GPA $\overline{G} = \langle\{\overline{V}, \overline{E}\}\rangle = \langle\{\}, \{\}\rangle$. Next, for each $G_{\pi_i} = \langle V_{\pi_i}, E_{\pi_i}\rangle \in \mathcal{T}$, we merge the concrete directed hypergraph with $\overline{G}$ as follows: $\overline{V} = \overline{V} \cup \{\alpha(F_\alpha, v) | v \in V_{\pi_i}\}$. Next, we form a set of abstract hyperedges by converting every edge in E_{π_i} to its corresponding abstract counterpart, i.e., $\overline{E}_{\pi_i} = \{\langle \overline{e}_{start}, \overline{e}_{results}, \overline{e}_{action}\rangle\}$ for every edge $e \in E_{\pi_i}$ s.t. $\overline{e}_{start} = \alpha(F_\alpha, e_{start})$, $\overline{e}_{results} = \{\alpha(F_\alpha, v) | v \in e_{results}\}$ and $\overline{e}_{action} = \beta(F_\beta, e_{start}, e_{action})$. Finally, we merge this set of abstract hyperedges using the following procedure: $\overline{E} = \overline{E} \cup \overline{E}_{\pi_i}$. Once this is done, we compress the GPA by replacing any edges in $\overline{E}$ that have the same start nodes and labels with a new edge, i.e., for any two edges $\overline{e}^1, \overline{e}^2 \in \overline{E}$ s.t. $\overline{e}^1_{start} = \overline{e}^2_{start}$, and $\overline{e}^1_{action} = \overline{e}^2_{action}$, we delete the two edges and replace the edge set with a new merged edge: $\overline{E} = \langle \overline{e}^1_{start}, \overline{e}^1_{results} \cup \overline{e}^2_{results}, \overline{e}^1_{action}\rangle \cup \overline{E} \setminus \{\overline{e}^1, \overline{e}^2\}$.

3.4 Solving SSPs using GPAs

Given a GPA $\overline{G}$, we now describe our method for solving SSPs. To do so, we modify the cost function of an SSP P to generate a new SSP $P|_{\overline{G}}$, taking into account the transitions in $\overline{G}$ as shown in Alg. 1 (lines 1-7). Our SSP solver -- GPA-accelerated SSP Solver -- operates as follows: Create a copy of the cost function of the original SSP P (line 1); Iterate over each (s, a, s') tuple of P and convert them to $(\overline{s}, \overline{a}, \overline{s'})$ tuples by computing abstract states and action representations using canonical abstraction (lines 2-3); Check if a hyperedge corresponding to the tuple $(\overline{s}, \overline{a}, \overline{s'})$ exists in the GPA $\overline{G}$. A hyperedge $(e_{start}, e_{results}, e_{action}) \in \overline{E}$ is corresponding to the tuple $(\overline{s}, \overline{a}, \overline{s'})$ iff $e_{start} = \overline{s}$, $\overline{s'} \in e_{results}$, and $e_{action} = \overline{a}$ (line 4); If the hyperedge does not exist in the GPA $\overline{G}$, we set the cost entry for the concrete tuple (s, a, s') to infinity (line 5). Finally, we create a GPA-constrained SSP $P|_{\overline{G}}$ that is same as the original SSP except for the cost function which is a modified copy of the original cost function (line 8), and use any off-the-shelf SSP solver that guarantees to find a partial proper policy if one exists to solve $P|_{\overline{G}}$ (line 9).

The goal of modifying the cost function is to prevent transitions whose abstract translations are not present in the Generalized Policy Automaton $\overline{G}$ to be used when performing Bellman updates for the new SSP $P|_{\overline{G}}$. As a result, actions belonging to such transitions cannot appear in $\pi_{P|_{\overline{G}}}$. As a consequence, the existence of a partial proper policy is not guaranteed in $P|_{\overline{G}}$. Nevertheless, the GPA-accelerated SSP solver is guaranteed to return a partial proper policy. This is because, if the computed policy for $P|_{\overline{G}}$ is a partial proper policy, the GPA-accelerated SSP solver directly returns it (lines 10-11). Otherwise, if a partial proper policy for $P|_{\overline{G}}$ does not exist or an improved policy is desired, then the GPA-accelerated SSP solver re-uses the computational effort expended in $P|_{\overline{G}}$ to initialize the value function of P using $v_{P|_{\overline{G}}}$ (line 13) and uses the SSP solver to compute a policy for the original SSP P (lines 14-15). This policy is guaranteed to be a partial proper policy if the SSP solver used is complete (Thm. 3.2). Also, if the used SSP solver is optimal then the bootstrapped policy computed for P is at least as cost-effective as the policy computed for $P|_{\overline{G}}$ (Thm. 3.1).

The GPA-accelerated SSP solver computes a policy for $P|_{\overline{G}}$ in the space of cross-product of the states of the GPA $\overline{G}$ with the states of the SSP P, similar to HAMs [Parr and Russell, 1997]. This policy, defined as a *hierarchically optimal policy*, corresponds to an optimal policy for P among all the policies that satisfy the constraints encoded by the GPA $\overline{G}$. As seen in our empirical analysis in Sec. 4, we observe that using a small set of example policies that are sufficient to capture rich generalized control structures results in GPA-accelerated SSP solver returning partial proper policies within a fraction of the original computational effort. We now prove theoretical guarantees of completeness and hierarchical optimality below.[2]

Theorem 3.1. *Given an SSP P, an optimal SSP solver, a GPA $\overline{G}$, and the corresponding GPA-constrained SSP $P|_{\overline{G}}$, let v_P, $v_{P|_{\overline{G}}}$, and π_P, $\pi_{P|_{\overline{G}}}$ be the state-value functions and policies obtained using Alg. 1 respectively, then π_P is at least as cost-effective as $\pi_{P|_{\overline{G}}}$, i.e., $v_P(s_0) \leq v_{P|_{\overline{G}}}(s_0)$.*

Proof (Sketch). Our proof is based on the following intuition. In the trivial case where $\pi_{P|_{\overline{G}}}$ is not a partial proper policy (line 10), $v_{P|_{\overline{G}}}(s_0) = \infty$ and solving P using bootstrapped estimates (line 13) is equivalent to solving P without bootstrapping. If $\pi_{P|_{\overline{G}}}$ is a partial proper policy, then it is also a partial proper policy for P since Alg. 1 does not change the transition function of P and thus $v_P(s_0) = v_{P|_{\overline{G}}}(s_0)$ when bootstrapping. An optimal solver using the Bellman equations will only accept a new policy π over $\pi_{P|_{\overline{G}}}$ if it leads to improvement over the *current, greedy* policy i.e., $\pi_{P|_{\overline{G}}}$. Thus, the only case in which Alg. 1 would return a different policy is if it finds a partial proper policy whose expected cost is less than or equal to that of $\pi_{P|_{\overline{G}}}$ i.e., $v_P(s_0) \leq v_{P|_{\overline{G}}}(s_0)$. □

Theorem 3.2. *Given a training set $\mathcal{T} = \{G_{\pi_1}, ..., G_{\pi_n}\}$ containing partial proper policies, an optimal, complete SSP solver, feature sets F_α, F_β, a learned GPA $\overline{G}$, and an SSP P, the GPA-accelerated SSP Solver (Alg. 1) is guaranteed to be complete, i.e., return a partial proper policy for P if one exists.*

Proof (Sketch). Line 10 of the Alg. 1 checks if the policy computed on solving $P|_{\overline{G}}$ is a partial proper policy and returns it in case it is partial proper policy. If the policy is not a partial proper policy then it uses the bootstrapped value estimates and solves P. If the used SSP solver is complete, then Alg. 1 is guaranteed to return a partial proper policy if one exists. □

Theorem 3.3. *Given a training set $\mathcal{T} = \{G_{\pi_1}, ..., G_{\pi_n}\}$ containing partial proper policies, an optimal complete SSP solver, feature sets F_α, F_β, a learned GPA $\overline{G}$, and an SSP P, the GPA-accelerated SSP Solver (Alg. 1) is guaranteed to return a hierarchically optimal policy for P if one exists.*

Proof (Sketch). We provide a proof by contradiction. Suppose that an optimal policy $\pi_* \neq \pi_{P|_{\overline{G}}}$ exists for $P|_{\overline{G}}$. This implies that for at least one of the states s in $P|_{\overline{G}}$, $v_*(s) < v_{\pi_{P|_{\overline{G}}}}(s)$ and as a result $\pi_*(s) \neq \pi_{P|_{\overline{G}}}(s)$. This implies the existence of an optimal action a for which the cost function $C'[s, a, s']$ was set to ∞ since otherwise this action a would have been considered by the SSP solver. This implies that a transition (s, a, s') in the training data was not captured by the GPA and is a contradiction since no transitions are discarded when creating $\overline{G}$ from the training data $\mathcal{T}$. □

[2]Complete proofs for all theorems are available in the appendix.

4 Experiments

We conducted an empirical evaluation on five well-known benchmark domains that were selected from the International Planning Competition (IPC) and International Probabilistic Planning Competition (IPPC) [Younes *et al.*, 2005] as well as robotic planning [Shah *et al.*, 2020]. As a part of our analysis, we aim to determine if GPAs allow efficient solving of SSPs.

We chose PPDDL as our representational language, which was the default language in IPPCs until 2011, after which, the Relational Dynamic Influence Language (RDDL) [Sanner, 2010] became the default. We chose PPDDL over RDDL since RDDL does not allow specifying the goal condition easily and as a result many benchmarks using RDDL are general purpose MDPs with no goals. Also, modern SOA solvers for PPDDL are available.

For our baselines, we focus on complete solvers for SSPs. We used Labeled RTDP (LRTDP) [Bonet and Geffner, 2003] and Soft-FLARES [Pineda and Zilberstein, 2019] which are state-of-art (SOA), complete SSP solvers. These algorithms internally generate their own heuristics using the domain and problem file. We conducted an empirical evaluation and used the FF heuristic [Hoffmann, 2001] as the internal heuristic function for all algorithms. We used FF even though it is inadmissible since the baselines performed best using it.

We ran our experiments on a cluster of Intel Xeon E5-2680 v4 CPUs running at 2.4 GHz with 16 GiB of RAM. Our implementation is in Python and we ported C++ implementations of the baselines from Pineda and Zilberstein [2019] to Python. We utilized problem generators from the IPPC suite and those in Shah *et al.* [2020] for generating the training and test problems for all domains. We provide a brief description of the problem domains below.

Rover(r, w, s, o) A set of r rovers need to collect and drop s samples that are present at one of w waypoints. The rovers also need to collect images of different objectives o that are visible from certain waypoints. This is an IPC domain and we converted it into a stochastic version by modifying sample collecting actions to fail with a probability of 0.4 (keeping the rover in the same state).

Gripper(b) A robot with two grippers is placed in an environment consisting of two rooms A and B. The objective of the robot is to transfer all the balls b initially located in room A to room B. We modified the gripper to be slippery so that picking balls have a 20% chance of failure.

Schedule(C, p) is an IPPC domain that consists of a set of p packets each belonging to one of C different classes that need to be queued. A router must first process the arrival of a packet in order to route it. The interval at which the router processes arrivals is determined by probability 0.94.

Keva(P, h) A robot uses P keva planks to build a tower of height h. Planks are placed in a specific order in one of the two locations preferring one location with probability 0.6. Despite this simple setting, the keva domain has been shown

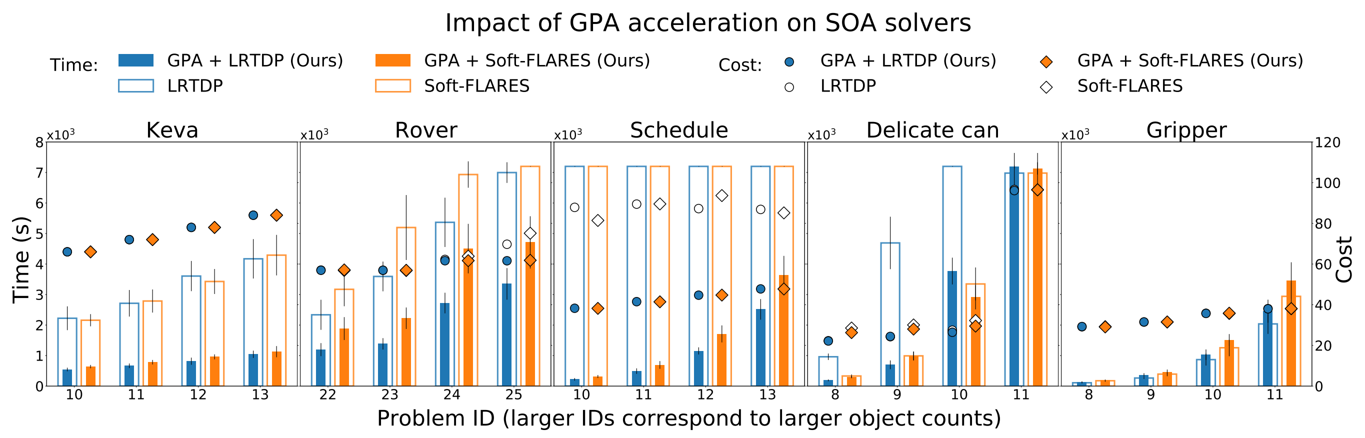

Figure 1: Impact of learned GPAs on solver performance (lower values are better). Left y-axes and bars show solution times (in units of 1000 secs) for our approach and baseline SOA solvers (LRTDP and Soft-FLARES). Right y-axes and points show cost incurred by the policy computed by our approach and the baselines. We use the same SSP solver as the corresponding baseline in our approach. Error bars indicate 1 standard deviation (SD) averaged across 10 runs. For clarity, we only report results for the largest test problems and omit error bars from costs due to the low SDs. This and additional empirical data is available in the supplementary material.

to be a challenging robotics problem [Shah *et al.*, 2020].

Delicate Can(c) An arrangement of c cans on a table of which 1 is a delicate can. The objective is to pick up a specific goal can. Cans can obstruct the trajectory to the goal can and they must be moved in order to successfully pick up the goal can. Cans can be crushed with probability 0.1 (delicate cans have a higher chance with probability 0.8) and need to be revived.

Training Setup Our method learns GPAs in a few-shot fashion, requiring little to no training data. For our training set, we used at most ten optimal solution policies (obtained using LAO*) for each domain. The time required to learn a GPA was less than 10 seconds in all cases in our experiments. This highlights the advantages of GPAs that can quickly be learned in a few-shot setting.

Test Setup We fixed the time and memory limit for each problem to 7200 seconds and 16GiB respectively. To demonstrate generalizability, our test set contains problems with object counts that are much larger than the training policies used. The largest problems in our test sets contain at least twice the number of objects than those used in training. For example, in the Keva domain we use training policies with towers of height up to 6 and evaluate on problems with towers of height up to 14. The minimum and maximum number of problems that we used in our test set are 11 and 26 problems respectively. Due to space limitations, information pertaining to the total number of training and test problems and their parameters, used hyperparameters for configuring baselines, etc., are included in the supplementary material.

4.1 Results and Analysis

Our evaluation metric compares the time required to find a partial proper policy. We also compare the quality of the computed policies by executing the policies for 100 trials with a horizon limit of 100 and averaging the obtained costs. We report our overall results averaged across 10 different runs and report results up to 1 standard deviation. Results of our experiments are illustrated in Fig. 1.

In four out of five domains (Schedule, Rover, Keva, and Delicate Can), our approach takes significantly less time compared to the corresponding baseline. For example, in Schedule, the baselines timed out for all of the large test problems reported. GPAs are able to successfully prune away action transitions that are not helping, leading to large savings in the computational effort. The costs obtained for executing these policies are also quite similar to each baseline showing that GPAs are capable of learning good policies much faster without compromising solution quality.

Our approach was unable to outperform the baselines in the Gripper domain. An interesting phenomenon that we observed was that training policies returned by LAO* were different for the case of even/odd balls due to tie-breaking and this led to the GPA not pruning actions as effectively. Nevertheless, we expected GPA to still outperform the baselines. We performed a deeper investigation and found that the FF heuristic used is already well-suited to prune away actions that the GPA would have otherwise pruned. This results in additional overhead being added in our SSP solver from the process of abstraction. However, heuristics that allow such pruning are difficult to synthesize and in many cases are hand-coded by a domain expert after employing significant effort.

Finally, because of the fixed timeout used, the maximum time of the baselines was bounded, making the impact of GPA appear smaller in larger problems. For example, in problem ID 25 of the Rover domain, the Soft-FLARES baseline timed out in all our runs, but when allowed to run to convergence, it took over 15000 seconds in a targeted experiment that we performed for investigating this issue.

5 Related Work

There has been plenty of dedicated research to improve the efficiency for solving SSPs. LAO* [Hansen and Zilberstein, 2001] computes policies by using heuristics to guide the search process. LRTDP [Bonet and Geffner, 2003] uses a labeling procedure in RTDP wherein a part of the sub-tree that is ϵ-consistent is marked as *solved* leading to faster ending of trials. SSiPP [Trevizan and Veloso, 2012] uses short-sightedness by only considering reachable states up to t states away and solving this constrained SSP. Soft-FLARES [Pineda and Zilberstein, 2019] combines labeling and short-sightedness for computing solutions. These approaches are complete and can be configured to return optimal solutions, however, they fail to learn any generalized knowledge and as result cannot readily scale to larger problems with a greater number of objects.

Boutilier *et al.* [2001] utilize decision-theoretic regression to compute generalized policies for first-order MDPs represented using situation calculus. They utilize symbolic dynamic programming to compute a symbolic value function that applies to problems with varying number of objects. FOALP [Sanner and Boutilier, 2005] uses linear programming to compute an approximation of the value function for first-order MDPs while providing upper bounds on the approximation error irrespective of the domain size. A key limitation of their approach is requiring the use of a representation of action models over which it is possible to regress using situation calculus. API [Fern *et al.*, 2006] uses approximate policy iteration with taxonomic decision lists to form policies. They use Monte Carlo simulations with random walks on a single problem to construct a policy. API offers no guarantees of completeness or hierarchical optimality.

Parr and Russell [1997] propose the hierarchical abstract machine (HAM) framework wherein component solutions from problem instances can be combined to solve larger problem instances efficiently. Recently, Bai and Russell [2017] extended HAMs to Reinforcement Learning settings by leveraging internal transitions of the HAMs. A key limitation of both these approaches is that the HAMs were hand-coded by a domain expert.

Bonet *et al.* [2009] automatically create finite-state controllers for solving problems using a set of examples by modeling the search as a contingent problem. Their approach is limited in applicability since it only works on deterministic problems and the features they use are hand-coded. Aguas *et al.* [2016] utilize small example policies to synthesize hierarchical finite state controllers that can call each other. However, their approach requires all training data to be provided upfront. D2L [Bonet *et al.*, 2019; Francès *et al.*, 2021] utilizes description logics to automatically generate features and reactive policies based on those features. Their approach comes with no guarantees for finding a solution or its cost and can only work on deterministic problems.

Our approach differs from these approaches in several aspects. Our approach constructs a GPA automatically without any human intervention. Using canonical abstraction, we lift problem-specific characteristics like object names and object counts. Another key difference between other techniques is that our approach can easily incorporate solutions from new examples into the GPA without having to remember any of the earlier examples. This allows our learning to scale better and can naturally utilize *leapfrogging* [Groshev *et al.*, 2018a; Karia and Srivastava, 2021] when presented with a large problem in the absence of training data. Finally, our approach comes with guarantees of completeness and hierarchical optimality given the training data presented. This means that if a solution exists, our approach will find it and it will be guaranteed to be hierarchically optimal.

6 Conclusions and Future Work

We show that non-deterministic Generalized Policy Automata (GPAs) constructed using solutions of small example SSPs are able to significantly reduce the computational effort for finding solutions for larger related SSPs. Furthermore, for many benchmark problems, the search space pruned by GPAs does not prune away relevant transitions allowing our approach to compute policies of comparable cost in a fraction of the effort. Our approach comes with guarantees of hierarchical optimality given the training data and also comes with the guarantee of always finding a solution provided one exists.

There are several interesting research directions for future work. Currently, GPAs do not employ any memory and can be improved by incorporating a finite amount of memory for better pruning of the action transitions. Description Logics (DL) are more expressive than canonical abstractions and have been demonstrated by Bonet *et al.* [2019] to be effective at synthesizing memoryless controllers for deterministic planning problems. Our approach can easily utilize any relational abstraction and it would be interesting to evaluate the efficacy of description logics as compared to canonical abstractions. Finally, our approach is applicable when solutions have a pattern. We believe that more intelligent training data generation methods could help improve performance in domains like Gripper. We plan to investigate these directions of research in future work.

Acknowledgements

We thank anonymous reviewers for their helpful feedback on the paper. This work was supported by the NSF under grants IIS 1909370 and IIS 1942856.

References

[Aguas *et al.*, 2016] Javier Segovia Aguas, Sergio Jiménez Celorrio, and Anders Jonsson. Hierarchical finite state controllers for generalized planning. In *IJCAI*, 2016.

[Bai and Russell, 2017] Aijun Bai and Stuart J. Russell. Efficient reinforcement learning with hierarchies of machines by leveraging internal transitions. In *IJCAI*, 2017.

[Bertsekas and Tsitsiklis, 1996] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

[Bonet and Geffner, 2003] Blai Bonet and Hector Geffner. Labeled RTDP: improving the convergence of real-time dynamic programming. In *ICAPS*, 2003.

[Bonet *et al.*, 2009] Blai Bonet, Héctor Palacios, and Hector Geffner. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS*, 2009.

[Bonet *et al.*, 2019] B. Bonet, G. Francès, and H. Geffner. Learning features and abstract actions for computing generalized plans. In *AAAI*, 2019.

[Boutilier *et al.*, 2001] C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order mdps. In *IJCAI*, 2001.

[Fern *et al.*, 2006] A. Fern, S. W. Yoon, and R. Givan. Approximate policy iteration with a policy language bias: Solving relational markov decision processes. *J. Artif. Intell. Res.*, 25:75–118, 2006.

[Francès *et al.*, 2021] G. Francès, B. Bonet, and H. Geffner. Learning general planning policies from small examples without supervision. In *AAAI*, 2021.

[Garg *et al.*, 2020] S. Garg, A. Bajpai, and Mausam. Symbolic network: Generalized neural policies for relational mdps. In *ICML*, 2020.

[Groshev *et al.*, 2018a] Edward Groshev, Maxwell Goldstein, Aviv Tamar, Siddharth Srivastava, and Pieter Abbeel. Learning generalized reactive policies using deep neural networks. In *ICAPS*, 2018.

[Groshev *et al.*, 2018b] Edward Groshev, Aviv Tamar, Maxwell Goldstein, Siddharth Srivastava, and Pieter Abbeel. Learning generalized reactive policies using deep neural networks. In *AAAI Spring Symposium Series*, 2018.

[Hansen and Zilberstein, 2001] Eric A Hansen and Shlomo Zilberstein. Lao: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence*, 129(1-2):35–62, 2001.

[Hoffmann, 2001] Jörg Hoffmann. FF: the fast-forward planning system. *AI Mag.*, 22(3):57–62, 2001.

[Karia and Srivastava, 2021] R. Karia and S. Srivastava. Learning generalized relational heuristic networks for model-agnostic planning. In *AAAI*, 2021.

[Kolobov *et al.*, 2012] Andrey Kolobov, Mausam, and Daniel S. Weld. A theory of goal-oriented mdps with dead ends. In *UAI*, 2012.

[Levy *et al.*, 2019] Andrew Levy, George Konidaris, Robert Platt, and Kate Saenko. Learning multi-level hierarchies with hindsight. In *ICLR*, 2019.

[Parr and Russell, 1997] Ronald Parr and Stuart J. Russell. Reinforcement learning with hierarchies of machines. In *NeurIPS*, 1997.

[Pineda and Zilberstein, 2019] Luis Enrique Pineda and Shlomo Zilberstein. Soft labeling in stochastic shortest path problems. In *AAMAS*, 2019.

[Pong *et al.*, 2018] Vitchyr Pong, Shixiang Gu, Murtaza Dalal, and Sergey Levine. Temporal difference models: Model-free deep rl for model-based control. In *ICLR*, 2018.

[Sagiv *et al.*, 2002] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[Sanner and Boutilier, 2005] S. Sanner and C. Boutilier. Approximate linear programming for first-order mdps. In *UAI*, 2005.

[Sanner, 2010] S. Sanner. Relational dynamic influence diagram language (RDDL): Language description. http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf, 2010.

[Shah *et al.*, 2020] Naman Shah, Deepak Kala Vasudevan, Kislay Kumar, Pranav Kamojjhala, and Siddharth Srivastava. Anytime integrated task and motion policies for stochastic environments. In *ICRA*, 2020.

[Srivastava *et al.*, 2011] S. Srivastava, N. Immerman, and S. Zilberstein. A new representation and associated algorithms for generalized planning. *Artif. Intell.*, 175(2):615–647, 2011.

[Sutton and Barto, 1998] R. S. Sutton and A. G. Barto. *Reinforcement learning - an introduction*. MIT Press, 1998.

[Tamar *et al.*, 2016] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. *In NeurIPS*, 29, 2016.

[Toyer *et al.*, 2018] Sam Toyer, Felipe W. Trevizan, Sylvie Thiébaux, and Lexing Xie. Action schema networks: Generalised policies with deep learning. In *AAAI*, 2018.

[Trevizan and Veloso, 2012] Felipe W. Trevizan and Manuela M. Veloso. Short-sighted stochastic shortest path problems. In *ICAPS*, 2012.

[Wu and Givan, 2007] J. Wu and R. Givan. Discovering relational domain features for probabilistic planning. In *ICAPS*, 2007.

[Younes *et al.*, 2005] Håkan L. S. Younes, Michael L. Littman, David Weissman, and John Asmuth. The first probabilistic track of the international planning competition. *J. Artif. Intell. Res.*, 24:851–887, 2005.

A Proofs for Theoretical Results

A.1 Proof for Theorem 3.1

Proof. (Case 1) Consider the case in Alg. 1 where $\pi_{P|_{\overline{G}}}$ is not a partial proper policy (line 10). This implies that there is at least one state in $\pi_{P|_{\overline{G}}}$ from which the goal is not reachable. Let s_{dead} be such a state and let s and a by any state and action such that $\pi_{P|_{\overline{G}}}(s) = a$ and $t(s, a, s_{dead}) > 0$. The state-value estimate for s can be written as:

$$\begin{aligned}
v_{\pi_{P|_{\overline{G}}}}(s) &= \sum_{s' \in S} t(s,a,s')[c(s,a,s') + v_{\pi_{P|_{\overline{G}}}}(s')] \\
&= \sum_{s' \in S \setminus s_{dead}} t(s,a,s')[c(s,a,s') + v_{\pi_{P|_{\overline{G}}}}(s')] \\
&\quad + t(s,a,s_{dead})[c(s,a,s_{dead}) + v_{\pi_{P|_{\overline{G}}}}(s_{dead})] \\
&= \sum_{s' \in S \setminus s_{dead}} t(s,a,s')[c(s,a,s') + v_{\pi_{P|_{\overline{G}}}}(s')] \\
&\quad + t(s,a,s_{dead})[c(s,a,s_{dead}) + \infty] \\
&= \infty
\end{aligned}$$

Naturally, since for a partial proper policy, s is always reachable from the initial state via some path, we can recursively apply the Bellman equation to obtain $v_{\pi_{P|_{\overline{G}}}}(s_0) = \infty$. For any other path reachable from s_0 through a state s' such that the path is proper, it follows that $v_{\pi_{P|_{\overline{G}}}}(s') \neq \infty$. When such values are bootstrapped, the SSP solver will always try to find a proper path or adjust the path from s_0 such that the dead end states are never encountered. Since at least one such policy is guaranteed to exist, an SSP solver will find it and it will be strictly better than an improper policy, i.e., $v_P(s_0) \leq v_{P|_{\overline{G}}}(s_0)$.

(Case 2): In this case, Alg. 1 computes a partial proper policy $\pi_{P|_{\overline{G}}}$ and initializes the value estimates for the problem P with those from $v_{P|_{\overline{G}}}$. At the initial iteration, $\pi_{P|_{\overline{G}}}$ is also a partial proper policy for the original SSP P since the transition function has not been modified by Alg. 1. Thus, we only need to show that using an SSP solver always results in policy improvement over the current policy if a better one exists. Suppose that for a given state s and action a, $\pi_{P|_{\overline{G}}}(s) = a$. Let an action $a' \neq a$ exist such that $c(s, a', s')$ was set to ∞ by Alg. 1. Furthermore, let $\pi_*(s) = a'$ for SSP P. This implies that $v_*(s) \leq v_{P|_{\overline{G}}}(s_0)$. The SSP solver would try policy improvement for this new transition using the Bellman optimality equation using the current estimate of the state $v(s')$:

$$v(s) = \min_{a \in A} \sum_{s' \in S} t(s,a,s')[c(s,a,s') + v(s')]$$

It is easy to see that an optimal solver (usually configured with an admissible heuristic), would initialize $v(s')$ with an admissible heuristic estimate of the expected cost if s' was not explored earlier or it would simply use the bootstrapped estimate. In either case, if the value of the state $v(s)$ for action a' was found to be better than the current policy $\pi_{P|_{\overline{G}}}$, the solver would update the current policy to use a' instead. These set of updates would recursively trickle to the initial state and would always result in a lower expected cost. Thus, $v_P(s_0) \leq v_{P|_{\overline{G}}}(s_0)$. □

A.2 Proof for Theorem 3.2

Proof. The only places where Alg. 1 returns is when it finds a partial proper policy using the GPA (line 10) or using an SSP solver with bootstrapped estimates to solve P and returning the found policy. Since Alg. 1 guarantees policy improvement (Thm. 3.1), once it finds a partial proper policy, all subsequent policies are guaranteed to be partial proper policies. In the trivial case, line 10 checks that the policy computed by solving $P|_{\overline{G}}$ is partial proper and returns it. If $\pi_{P|_{\overline{G}}}$ is not a partial proper policy, then bootstrapped estimates are used for policy improvement for the original SSP where the existence of a partial proper policy is guaranteed. If the used SSP solver is complete, it is guaranteed to find such a policy. □

A.3 Proof for Theorem 3.3

Proof. (Case 1) Let $\mathcal{T} = \{\}$ in effect making the GPA $\overline{G} = \langle\{\},\{\}\rangle$ contain no transitions. Lines 2-6 of Alg. 1 will then mark every (s, a, s') tuple of the SSP with a cost of ∞ when synthesizing $P|_{\overline{G}}$. The policy $\pi_{P|_{\overline{G}}}$ for $P|_{\overline{G}}$ will be an improper policy and in fact this policy is a hierarchically optimal policy for P under the given constraints.

(Case 2) For the second case, let the training data $\mathcal{T}$ consist of all possible policies for a given SSP P. As a result, every transition that is represented by P should appear in the GPA. In this case, Alg. 1 will never modify the cost function and as a result $C' = C$. Since the transition function is not modified by Alg. 1, solving $P|_{\overline{G}}$ is equivalent to solving P and using an optimal solver would find the optimal policy π_* for P. It is easy to see that π_* is a hierarchically optimal policy given such a GPA. If all abstract transitions for a given domain appear in the GPA, Alg. 1 will find an optimal policy for any SSP P for such a domain.

(Case 3) Finally, we consider a training set $\mathcal{T} = \{G_{\pi_1}, ..., G_{\pi_n}\}$ that consists of policies from solving SSP problems. Consider a proper policy $\pi_{P|_{\overline{G}}}$ that is obtained by using an optimal solver to solve $P|_{\overline{G}}$ using Alg. 1. The value of any state $\pi_{P|_{\overline{G}}}$ when using $\pi_{P|_{\overline{G}}}$ can be expressed as:

$$v_{\pi_{P|_{\overline{G}}}}(s) = \min_{a \in A} \sum_{s' \in S} t(s,a,s')[c(s,a,s') + v_{\pi_{P|_{\overline{G}}}}(s')] \quad (1)$$

If there exists two different policies $\pi'_{P|_{\overline{G}}} \neq \pi_{P|_{\overline{G}}}$ s.t. $\pi'_{P|_{\overline{G}}}$ is strictly better than $\pi_{P|_{\overline{G}}}$ for a state s then this must mean that $\pi'_{P|_{\overline{G}}}(s) \neq \pi_{P|_{\overline{G}}}(s)$. Let $\pi'_{P|_{\overline{G}}}(s) = a$ and $\pi_{P|_{\overline{G}}} = a'$.

$$v_{\pi_{P|_{\overline{G}}}}(s) > v_{\pi'_{P|_{\overline{G}}}}(s) \quad (2)$$

$$\begin{aligned}
&\sum_{s' \in S} t(s,a,s')[c(s,a,s') + v(s')] > \\
&\qquad \sum_{s' \in S} t(s,a',s')[c(s,a',s') + v(s')]
\end{aligned} \quad (3)$$

If the transition (s, a', s') was present in the training data $\mathcal{T}$ then such a transition would appear in the GPA and as a result this policy would be found by the optimal SSP solver. Now suppose that a transition (s, a', s'') where $t(s, a', s'') > 0$ does not appear in the training data. Furthermore, suppose that no abstract hyperedge with $\overline{e}_{start} = s$ and $\overline{e}_{action} = a$ contains $\overline{s}''$ in $\overline{e}_{results}$ in $\overline{G}$. This would cause the check for the

	θ				
ID	Keva(P,h)	Rover(r,w,s,o)	Schedule(C,p)	Delicate Can(c)	Gripper(b)
0	$(2,1)$	$(1,3,1,2)$	$(1,2)$	(2)	(1)
1	$(4,2)$	$(1,4,1,2)$	$(1,3)$	(3)	(1)
2	$(6,3)$	$(1,3,2,2)$	$(1,4)$	(4)	(2)
3	$(8,4)$	$(1,4,2,2)$	–	(5)	(2)
4	$(10,5)$	$(1,3,3,2)$	–	(6)	(3)
5	$(12,6)$	$(1,4,3,2)$	–	–	(3)
6	–	$(1,3,4,2)$	–	–	(4)
7	–	$(1,4,4,2)$	–	–	(4)
8	–	$(1,3,5,2)$	–	–	(5)
9	–	$(1,4,5,2)$	–	–	(5)

Table 1: Our training setup for all domains. ID refers to the problem ID in the training set. The other columns refer to the parameters passed to the problem generator for generating the problem. Entries marked – indicate that there was no such problem in training set.

abstract hyperedge for the transition (s, a', s'') to fail in line 4 of Alg. 1. Thus, $C'[s, a', s'']$ would be set to ∞ for this transition. Eqn. 3 can be rewritten as:

$$\sum_{s'\in S} t(s,a,s')[c(s,a,s')+v(s')] > \sum_{s'\in S\setminus s''} t(s,a',s')[c(s,a',s')+v(s')] + t(s,a',s'')[c(s,a',s'')+v(s'')] \tag{4}$$

$$\sum_{s'\in S} t(s,a,s')[c(s,a,s')+v(s')] > \sum_{s'\in S\setminus s''} t(s,a',s')[c(s,a',s')+v(s')] + t(s,a',s'')[\infty+v(s'')] \tag{5}$$

$$\sum_{s'\in S} t(s,a,s')[c(s,a,s')+v(s')] > \infty \tag{6}$$

If we assume $\pi_{P|_{\overline{G}}}$ to be a partial proper policy, $\sum_{s'\in S} t(s,a,s')[c(s,a,s')+v(s')] < \infty$. This is a contradiction and any such policy $\pi'_{P|_{\overline{G}}}$ is strictly worse than $\pi_{P|_{\overline{G}}}$. Next, if we assume $\pi_{P|_{\overline{G}}}$ to be improper, then $v_{\pi_{P|_{\overline{G}}}}(s_0) = \infty$. Substituting $s = s_0$ and $a = \pi_{P|_{\overline{G}}}(s_0)$ in Eqn. 6 we get another contradiction since in this case $\pi_{P|_{\overline{G}}}(s_0)$ is as good as $\pi'_{P|_{\overline{G}}}(s_0)$. Thus, $\pi_{P|_{\overline{G}}}$ is a hierarchically optimal policy given the GPA $\overline{G}$ constructed using the training data $\mathcal{T}$. □

B Extended Experiments and Results

Training and Test Setup Descriptions of the training problems used by us and their parameters can be found in Table. 1. Test problems and parameters along with complete information for the solution times, costs, and their standard deviations for 10 runs are available in tabular format in Tables 2, 3, 4, 5, and 6. Note that for the Keva domain, the standard deviations for the costs incurred are 0. This is accurate since the only source of stochasticity in Keva is a *human place* action that determines where the human places a plank which is one of two locations. As a result, Keva policies are deterministic in execution since the human always places a plank and all other actions are deterministic. It is interesting that despite

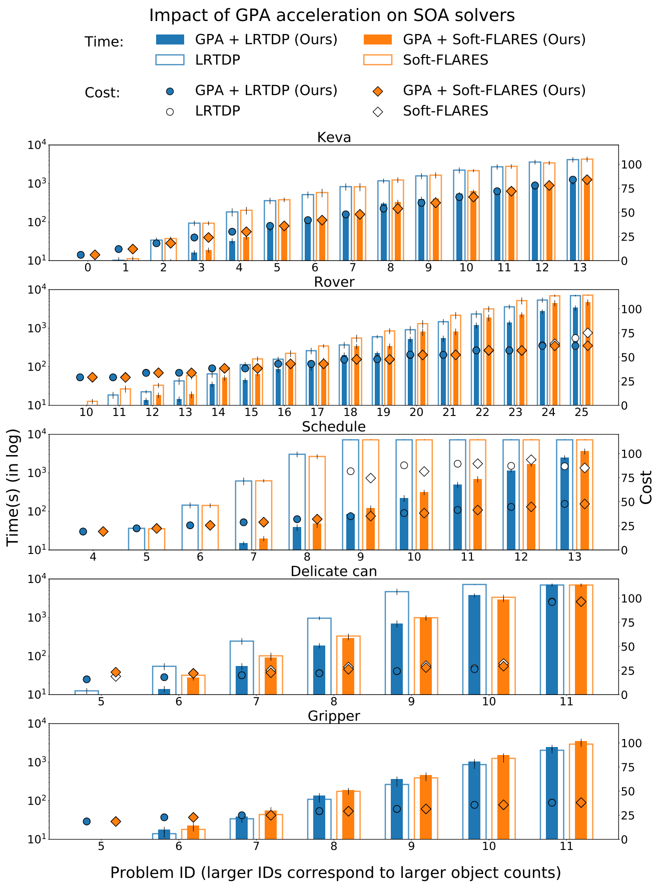

Figure 2: Impact of learned GPAs on solver performance of all test problems (lower values are better). Left y-axes and bars show solution times (in log scale), and right y-axes and points show cost incurred by the policy computed by our approach and baseline SOA solvers (LRTDP and Soft-FLARES). We use the same SSP solver as the corresponding baseline in our approach. Error bars for solution times indicate 1 standard deviation (SD) averaged across 10 runs.

this simplistic domain, the baselines are unable to reasonably converge within the timeout. We also present an extended version of Fig. 1 of the main paper that includes results for a larger suite a test problems for a better view of our overall results. These results are reported in Fig. 2. Note that the solution times on the left y-axis of these plots are shown in log scale. We omitted the problems with smaller IDs, mainly whose solutions times (in log scale) were not visible for both our as well as baseline approaches, for better visualization.

Hyperparameters We used $\epsilon = 10^{-5}$ as the value for determining whether an algorithm has converged to an ϵ-consistent policy. We set the total number of trials for all algorithms to ∞. As a result, each algorithm would only return once it has converged or if the time limit has been exceeded. For Soft-FLARES, we used $t = 4$ which controls the horizon of the sub-tree that is checked for being ϵ-consistent during the labeling procedure. Our distance metric is the *step* function which simply counts the depth until the horizon is exceeded. For the selective sampling procedure, we used the *logistic* sampler configured with $\alpha = 0.1$ and $\beta = 0.9$.

ID	θ	Time($x \equiv$ LRTDP)		Time($x \equiv$ Soft-FLARES)		Cost($x \equiv$ LRTDP)		Cost($x \equiv$ Soft-FLARES)	
		x	Ours + x	x	Ours + x	x	Ours + x	x	Ours + x
0	$(29, 1)$	2.45 $\pm$0.32	**1.44** $\pm$0.17	2.65 $\pm$0.37	**1.47** $\pm$0.27	6.00 $\pm$0.00	6.00 $\pm$0.00	6.00 $\pm$0.00	6.00 $\pm$0.00
1	$(29, 2)$	10.23 $\pm$2.05	**4.15** $\pm$0.68	11.22 $\pm$1.45	**4.63** $\pm$0.58	12.00 $\pm$0.00	12.00 $\pm$0.00	12.00 $\pm$0.00	12.00 $\pm$0.00
2	$(29, 3)$	34.05 $\pm$5.53	**8.14** $\pm$0.92	37.33 $\pm$4.71	**9.55** $\pm$1.45	18.00 $\pm$0.00	18.00 $\pm$0.00	18.00 $\pm$0.00	18.00 $\pm$0.00
3	$(29, 4)$	93.78 $\pm$14.53	**16.26** $\pm$2.34	93.39 $\pm$11.30	**18.80** $\pm$2.84	24.00 $\pm$0.00	24.00 $\pm$0.00	24.00 $\pm$0.00	24.00 $\pm$0.00
4	$(29, 5)$	183.57 $\pm$43.80	**32.61** $\pm$5.40	201.99 $\pm$45.05	**41.19** $\pm$6.05	30.00 $\pm$0.00	30.00 $\pm$0.00	30.00 $\pm$0.00	30.00 $\pm$0.00
5	$(29, 6)$	358.98 $\pm$67.32	**68.73** $\pm$11.04	379.84 $\pm$51.43	**74.47** $\pm$9.10	36.00 $\pm$0.00	36.00 $\pm$0.00	36.00 $\pm$0.00	36.00 $\pm$0.00
6	$(29, 7)$	515.85 $\pm$101.56	**115.28** $\pm$11.90	583.52 $\pm$143.15	**120.30** $\pm$17.58	42.00 $\pm$0.00	42.00 $\pm$0.00	42.00 $\pm$0.00	42.00 $\pm$0.00
7	$(29, 8)$	829.19 $\pm$174.46	**174.88** $\pm$17.36	825.84 $\pm$192.54	**191.44** $\pm$27.16	48.00 $\pm$0.00	48.00 $\pm$0.00	48.00 $\pm$0.00	48.00 $\pm$0.00
8	$(29, 9)$	1174.79 $\pm$160.45	**298.61** $\pm$30.82	1236.91 $\pm$200.02	**331.95** $\pm$45.28	54.00 $\pm$0.00	54.00 $\pm$0.00	54.00 $\pm$0.00	54.00 $\pm$0.00
9	$(29, 10)$	1578.38 $\pm$279.31	**394.71** $\pm$62.16	1647.76 $\pm$297.42	**406.45** $\pm$37.23	60.00 $\pm$0.00	60.00 $\pm$0.00	60.00 $\pm$0.00	60.00 $\pm$0.00
10	$(29, 11)$	2223.43 $\pm$390.05	**544.26** $\pm$58.84	2158.91 $\pm$199.54	**639.60** $\pm$54.35	66.00 $\pm$0.00	66.00 $\pm$0.00	66.00 $\pm$0.00	66.00 $\pm$0.00
11	$(29, 12)$	2713.78 $\pm$435.39	**665.29** $\pm$76.02	2787.92 $\pm$378.85	**782.82** $\pm$74.38	72.00 $\pm$0.00	72.00 $\pm$0.00	72.00 $\pm$0.00	72.00 $\pm$0.00
12	$(29, 13)$	3606.12 $\pm$494.69	**815.11** $\pm$117.62	3427.93 $\pm$409.34	**958.17** $\pm$81.59	78.00 $\pm$0.00	78.00 $\pm$0.00	78.00 $\pm$0.00	78.00 $\pm$0.00
13	$(29, 14)$	4171.27 $\pm$645.70	**1042.44** $\pm$119.63	4291.64 $\pm$663.83	**1128.84** $\pm$183.24	84.00 $\pm$0.00	84.00 $\pm$0.00	84.00 $\pm$0.00	84.00 $\pm$0.00

Table 2: Our test setup for the Keva(P, h) domain (lower values better). ID refers to the problem ID in the test set. θ refers to the parameters passed to the problem generator for generating the problem. Times indicate the seconds required to find a policy. Similarly, costs are reported as average costs obtained by executing the computed policy for 100 trials. We ran our experiments using a different random seed for 10 different runs and report average metrics up to one standard deviation. Better metrics are at least 5% better and are indicated using bold font.

ID	θ	Time($x \equiv$ LRTDP)		Time($x \equiv$ Soft-FLARES)		Cost($x \equiv$ LRTDP)		Cost($x \equiv$ Soft-FLARES)	
		x	Ours + x	x	Ours + x	x	Ours + x	x	Ours + x
0	$(1, 3, 1, 2)$	0.02 $\pm$0.01	0.02 $\pm$0.01	0.02 $\pm$0.01	**0.01** $\pm$0.00	6.62 $\pm$0.08	6.68 $\pm$0.08	6.69 $\pm$0.12	6.68 $\pm$0.09
1	$(1, 4, 1, 2)$	0.02 $\pm$0.01	0.02 $\pm$0.00	0.02 $\pm$0.01	**0.01** $\pm$0.00	6.64 $\pm$0.08	6.70 $\pm$0.09	6.62 $\pm$0.12	6.70 $\pm$0.12
2	$(1, 3, 2, 2)$	0.10 $\pm$0.03	**0.03** $\pm$0.01	0.09 $\pm$0.02	**0.03** $\pm$0.00	10.33 $\pm$0.14	10.29 $\pm$0.08	10.38 $\pm$0.15	10.34 $\pm$0.10
3	$(1, 4, 2, 2)$	0.18 $\pm$0.04	**0.03** $\pm$0.01	0.22 $\pm$0.02	**0.04** $\pm$0.01	10.34 $\pm$0.11	10.30 $\pm$0.12	10.27 $\pm$0.12	10.26 $\pm$0.16
4	$(1, 3, 3, 2)$	0.43 $\pm$0.06	**0.18** $\pm$0.03	0.46 $\pm$0.07	**0.20** $\pm$0.04	15.03 $\pm$0.19	15.01 $\pm$0.16	14.94 $\pm$0.23	15.09 $\pm$0.11
5	$(1, 4, 3, 2)$	0.80 $\pm$0.14	**0.10** $\pm$0.02	0.96 $\pm$0.24	**0.09** $\pm$0.01	15.00 $\pm$0.23	15.01 $\pm$0.23	14.95 $\pm$0.17	14.95 $\pm$0.19
6	$(1, 3, 4, 2)$	1.08 $\pm$0.12	**0.54** $\pm$0.10	1.66 $\pm$0.39	**0.68** $\pm$0.11	19.76 $\pm$0.21	19.65 $\pm$0.31	19.68 $\pm$0.15	19.62 $\pm$0.17
7	$(1, 4, 4, 2)$	2.33 $\pm$0.43	**0.54** $\pm$0.08	3.22 $\pm$0.69	**0.70** $\pm$0.14	19.63 $\pm$0.13	19.63 $\pm$0.18	19.70 $\pm$0.16	19.73 $\pm$0.20
8	$(1, 3, 5, 2)$	3.54 $\pm$0.59	**1.61** $\pm$0.23	4.25 $\pm$0.84	**2.42** $\pm$0.51	24.36 $\pm$0.17	24.35 $\pm$0.30	24.34 $\pm$0.22	24.46 $\pm$0.09
9	$(1, 4, 5, 2)$	7.78 $\pm$1.62	**1.82** $\pm$0.28	9.57 $\pm$1.69	**2.26** $\pm$0.42	24.41 $\pm$0.27	24.38 $\pm$0.16	24.32 $\pm$0.34	24.23 $\pm$0.32
10	$(1, 3, 6, 2)$	8.77 $\pm$1.08	**4.91** $\pm$1.04	12.67 $\pm$1.91	**6.40** $\pm$0.93	28.95 $\pm$0.26	29.13 $\pm$0.26	28.98 $\pm$0.25	29.14 $\pm$0.31
11	$(1, 4, 6, 2)$	18.50 $\pm$3.76	**4.31** $\pm$0.77	26.62 $\pm$5.28	**5.28** $\pm$0.56	28.96 $\pm$0.29	29.07 $\pm$0.18	29.09 $\pm$0.18	28.86 $\pm$0.25
12	$(1, 3, 7, 2)$	22.42 $\pm$2.26	**13.60** $\pm$1.98	33.11 $\pm$4.79	**18.49** $\pm$2.85	33.80 $\pm$0.29	33.54 $\pm$0.20	33.74 $\pm$0.26	33.71 $\pm$0.24
13	$(1, 4, 7, 2)$	42.96 $\pm$9.15	**14.47** $\pm$2.40	59.22 $\pm$12.54	**19.09** $\pm$3.84	33.70 $\pm$0.34	33.75 $\pm$0.22	33.63 $\pm$0.21	33.67 $\pm$0.28
14	$(1, 3, 8, 2)$	65.32 $\pm$13.63	**35.25** $\pm$5.75	93.44 $\pm$13.87	**51.72** $\pm$8.90	38.30 $\pm$0.26	38.39 $\pm$0.24	38.28 $\pm$0.27	38.36 $\pm$0.35
15	$(1, 4, 8, 2)$	113.36 $\pm$17.49	**44.60** $\pm$6.77	159.83 $\pm$25.30	**64.87** $\pm$7.88	38.43 $\pm$0.17	38.18 $\pm$0.22	38.33 $\pm$0.29	38.24 $\pm$0.36
16	$(1, 3, 9, 2)$	156.88 $\pm$23.55	**86.26** $\pm$16.11	222.42 $\pm$43.47	**122.44** $\pm$21.53	43.02 $\pm$0.40	42.98 $\pm$0.45	42.87 $\pm$0.26	43.11 $\pm$0.29
17	$(1, 4, 9, 2)$	260.78 $\pm$49.61	**95.92** $\pm$16.68	345.68 $\pm$38.97	**142.71** $\pm$27.75	43.05 $\pm$0.19	42.94 $\pm$0.31	42.98 $\pm$0.36	43.00 $\pm$0.24
18	$(1, 3, 10, 2)$	367.77 $\pm$71.46	**199.80** $\pm$35.50	555.65 $\pm$121.90	**337.10** $\pm$52.84	47.65 $\pm$0.34	47.36 $\pm$0.16	47.62 $\pm$0.30	47.67 $\pm$0.42
19	$(1, 4, 10, 2)$	599.74 $\pm$60.23	**223.38** $\pm$26.79	848.57 $\pm$141.78	**345.02** $\pm$53.64	47.85 $\pm$0.39	47.78 $\pm$0.27	47.49 $\pm$0.28	47.70 $\pm$0.19
20	$(1, 3, 11, 2)$	914.39 $\pm$217.86	**515.34** $\pm$85.61	1312.81 $\pm$302.78	**800.94** $\pm$169.25	52.17 $\pm$0.22	52.58 $\pm$0.35	52.29 $\pm$0.32	52.21 $\pm$0.29
21	$(1, 4, 11, 2)$	1472.73 $\pm$254.76	**543.15** $\pm$97.44	2168.73 $\pm$450.03	**819.44** $\pm$159.47	52.36 $\pm$0.27	52.35 $\pm$0.41	52.37 $\pm$0.46	52.26 $\pm$0.30
22	$(1, 3, 12, 2)$	2336.52 $\pm$492.69	**1195.16** $\pm$213.55	3171.35 $\pm$554.37	**1885.02** $\pm$375.15	57.00 $\pm$0.35	56.94 $\pm$0.26	56.85 $\pm$0.36	57.12 $\pm$0.62
23	$(1, 4, 12, 2)$	3593.42 $\pm$487.19	**1385.19** $\pm$187.00	5196.66 $\pm$1063.94	**2224.24** $\pm$351.60	56.93 $\pm$0.24	56.94 $\pm$0.24	56.80 $\pm$0.27	56.94 $\pm$0.35
24	$(1, 3, 13, 2)$	5366.69 $\pm$807.68	**2721.86** $\pm$337.95	6933.16 $\pm$432.32	**4505.52** $\pm$811.02	62.26 $\pm$1.69	61.63 $\pm$0.39	63.83 $\pm$2.43	61.67 $\pm$0.29
25	$(1, 4, 13, 2)$	6997.37 $\pm$338.99	**3349.16** $\pm$517.48	7200.00 $\pm$0.00	**4710.61** $\pm$855.90	69.72 $\pm$11.30	**61.60** $\pm$0.46	75.14 $\pm$13.01	**61.83** $\pm$0.37

Table 3: Our test setup for the Rover(r, w, s, o) domain (lower values better). ID refers to the problem ID in the test set. θ refers to the parameters passed to the problem generator for generating the problem. Times indicate the seconds required to find a policy. Similarly, costs are reported as average costs obtained by executing the computed policy for 100 trials. We ran our experiments using a different random seed for 10 different runs and report average metrics up to one standard deviation. Better metrics are at least 5% better and are indicated using bold font.

ID	θ	Time($x \equiv$ LRTDP)		Time($x \equiv$ Soft-FLARES)		Cost($x \equiv$ LRTDP)		Cost($x \equiv$ Soft-FLARES)	
		x	Ours + x	x	Ours + x	x	Ours + x	x	Ours + x
0	$(1, 2)$	0.02 $\pm$0.01	0.02 $\pm$0.01	0.02 $\pm$0.01	0.02 $\pm$0.01	6.41 $\pm$0.13	6.32 $\pm$0.13	6.35 $\pm$0.08	6.45 $\pm$0.15
1	$(1, 3)$	0.08 $\pm$0.02	**0.06** $\pm$0.02	0.07 $\pm$0.02	0.07 $\pm$0.02	9.53 $\pm$0.12	9.50 $\pm$0.09	9.60 $\pm$0.13	9.52 $\pm$0.12
2	$(1, 4)$	0.32 $\pm$0.05	**0.16** $\pm$0.03	0.33 $\pm$0.05	**0.19** $\pm$0.03	12.72 $\pm$0.14	12.73 $\pm$0.14	12.81 $\pm$0.15	12.77 $\pm$0.15
3	$(1, 5)$	1.58 $\pm$0.25	**0.41** $\pm$0.06	1.60 $\pm$0.34	**0.46** $\pm$0.09	16.00 $\pm$0.14	15.97 $\pm$0.17	15.89 $\pm$0.11	15.95 $\pm$0.19
4	$(1, 6)$	6.45 $\pm$0.77	**1.02** $\pm$0.22	7.36 $\pm$1.28	**1.21** $\pm$0.20	19.06 $\pm$0.16	19.16 $\pm$0.16	19.17 $\pm$0.24	19.14 $\pm$0.14
5	$(1, 7)$	36.46 $\pm$7.19	**2.46** $\pm$0.56	35.61 $\pm$6.35	**3.10** $\pm$0.59	22.37 $\pm$0.19	22.45 $\pm$0.19	22.45 $\pm$0.25	22.28 $\pm$0.24
6	$(1, 8)$	145.33 $\pm$24.97	**6.58** $\pm$1.25	142.70 $\pm$18.86	**8.42** $\pm$1.93	25.57 $\pm$0.18	25.56 $\pm$0.09	25.52 $\pm$0.23	25.59 $\pm$0.26
7	$(1, 9)$	616.36 $\pm$140.89	**14.92** $\pm$1.65	622.93 $\pm$61.96	**19.85** $\pm$3.00	28.78 $\pm$0.18	28.61 $\pm$0.16	28.73 $\pm$0.25	28.88 $\pm$0.18
8	$(1, 10)$	3036.01 $\pm$507.41	**38.89** $\pm$7.41	2662.96 $\pm$361.97	**48.08** $\pm$10.54	31.95 $\pm$0.08	31.95 $\pm$0.23	31.96 $\pm$0.20	32.01 $\pm$0.21
9	$(1, 11)$	7200.00 $\pm$0.00	**85.99** $\pm$9.94	7200.00 $\pm$0.00	**122.28** $\pm$19.92	81.68 $\pm$16.65	**35.06** $\pm$0.29	74.56 $\pm$19.98	**35.10** $\pm$0.28
10	$(1, 12)$	7200.00 $\pm$0.00	**220.64** $\pm$43.34	7200.00 $\pm$0.00	**313.88** $\pm$48.81	87.85 $\pm$11.06	**38.26** $\pm$0.23	81.42 $\pm$16.54	**38.20** $\pm$0.25
11	$(1, 13)$	7200.00 $\pm$0.00	**490.90** $\pm$90.83	7200.00 $\pm$0.00	**692.75** $\pm$126.60	89.45 $\pm$12.88	**41.50** $\pm$0.40	89.54 $\pm$11.68	**41.43** $\pm$0.22
12	$(1, 14)$	7200.00 $\pm$0.00	**1153.32** $\pm$117.37	7200.00 $\pm$0.00	**1710.56** $\pm$278.99	87.26 $\pm$12.32	**44.69** $\pm$0.13	93.66 $\pm$10.10	**44.74** $\pm$0.27
13	$(1, 15)$	7200.00 $\pm$0.00	**2514.16** $\pm$337.59	7200.00 $\pm$0.00	**3639.92** $\pm$627.08	86.84 $\pm$11.41	**47.77** $\pm$0.17	85.14 $\pm$13.60	**47.74** $\pm$0.34

Table 4: Our test setup for the Schedule(C, p) domain (lower values better). ID refers to the problem ID in the test set. θ refers to the parameters passed to the problem generator for generating the problem. Times indicate the seconds required to find a policy. Similarly, costs are reported as average costs obtained by executing the computed policy for 100 trials. We ran our experiments using a different random seed for 10 different runs and report average metrics up to one standard deviation. Better metrics are at least 5% better and are indicated using bold font.

ID	θ	Time($x \equiv$ LRTDP)		Time($x \equiv$ Soft-FLARES)		Cost($x \equiv$ LRTDP)		Cost($x \equiv$ Soft-FLARES)	
		x	Ours + x	x	Ours + x	x	Ours + x	x	Ours + x
0	(2)	0.01 $\pm$0.01	0.01 $\pm$0.00	**0.00** $\pm$0.00	0.01 $\pm$0.01	5.40 $\pm$0.07	5.40 $\pm$0.08	5.45 $\pm$0.07	5.40 $\pm$0.10
1	(3)	0.03 $\pm$0.01	**0.02** $\pm$0.01	0.03 $\pm$0.01	**0.02** $\pm$0.01	7.42 $\pm$0.12	7.50 $\pm$0.15	7.53 $\pm$0.09	7.48 $\pm$0.06
2	(4)	0.11 $\pm$0.03	**0.07** $\pm$0.02	0.12 $\pm$0.03	**0.09** $\pm$0.02	9.58 $\pm$0.11	9.53 $\pm$0.16	9.56 $\pm$0.14	9.62 $\pm$0.09
3	(5)	0.49 $\pm$0.08	**0.22** $\pm$0.05	0.51 $\pm$0.08	**0.40** $\pm$0.06	11.71 $\pm$0.20	11.72 $\pm$0.12	**12.48** $\pm$0.69	14.22 $\pm$2.49
4	(6)	2.56 $\pm$0.55	**0.80** $\pm$0.13	2.14 $\pm$0.26	**1.79** $\pm$0.23	13.82 $\pm$0.15	13.75 $\pm$0.07	**15.02** $\pm$1.09	25.16 $\pm$4.14
5	(7)	12.55 $\pm$2.20	**3.25** $\pm$0.56	8.08 $\pm$0.92	**6.81** $\pm$1.41	15.94 $\pm$0.09	15.86 $\pm$0.08	**18.87** $\pm$1.44	23.32 $\pm$4.09
6	(8)	54.31 $\pm$11.04	**14.12** $\pm$2.60	31.86 $\pm$5.15	**27.65** $\pm$4.41	18.08 $\pm$0.13	18.07 $\pm$0.14	22.16 $\pm$1.68	22.01 $\pm$1.52
7	(9)	244.32 $\pm$47.46	**54.60** $\pm$10.68	101.63 $\pm$20.29	**90.57** $\pm$15.57	20.04 $\pm$0.13	20.15 $\pm$0.14	25.18 $\pm$2.65	**22.70** $\pm$1.22
8	(10)	960.96 $\pm$105.78	**188.50** $\pm$29.21	331.06 $\pm$49.98	**293.54** $\pm$36.06	22.28 $\pm$0.17	22.17 $\pm$0.13	28.54 $\pm$2.09	**26.29** $\pm$1.73
9	(11)	4691.23 $\pm$859.08	**698.09** $\pm$143.63	990.65 $\pm$144.48	983.51 $\pm$144.08	24.30 $\pm$0.12	24.41 $\pm$0.17	30.05 $\pm$1.37	**28.02** $\pm$1.40
10	(12)	7200.00 $\pm$0.00	**3769.57** $\pm$439.21	3347.02 $\pm$540.66	**2911.30** $\pm$397.41	27.46 $\pm$0.83	26.46 $\pm$0.14	32.23 $\pm$2.35	**29.49** $\pm$1.47
11	(12)	6980.00 $\pm$660.00	7183.58 $\pm$49.27	6980.48 $\pm$658.56	7132.65 $\pm$202.04	96.62 $\pm$10.13	96.12 $\pm$11.65	96.50 $\pm$10.49	96.48 $\pm$10.56

Table 5: Our test setup for the Delicate Can(c) domain (lower values better). ID refers to the problem ID in the test set. θ refers to the parameters passed to the problem generator for generating the problem. Times indicate the seconds required to find a policy. Similarly, costs are reported as average costs obtained by executing the computed policy for 100 trials. We ran our experiments using a different random seed for 10 different runs and report average metrics up to one standard deviation. Better metrics are at least 5% better and are indicated using bold font.

ID	θ	Time($x \equiv$ LRTDP)		Time($x \equiv$ Soft-FLARES)		Cost($x \equiv$ LRTDP)		Cost($x \equiv$ Soft-FLARES)	
		x	Ours + x	x	Ours + x	x	Ours + x	x	Ours + x
0	(1)	0.01 $\pm$0.01	**0.00** $\pm$0.01	**0.00** $\pm$0.00	0.01 $\pm$0.01	3.25 $\pm$0.06	3.25 $\pm$0.05	3.23 $\pm$0.06	3.25 $\pm$0.04
1	(2)	0.02 $\pm$0.01	0.02 $\pm$0.01	0.02 $\pm$0.01	0.02 $\pm$0.01	5.48 $\pm$0.08	5.53 $\pm$0.08	5.51 $\pm$0.10	5.55 $\pm$0.07
2	(3)	**0.10** $\pm$0.02	0.12 $\pm$0.03	0.12 $\pm$0.03	0.12 $\pm$0.03	9.71 $\pm$0.07	9.76 $\pm$0.10	9.76 $\pm$0.07	9.75 $\pm$0.09
3	(4)	0.33 $\pm$0.05	0.34 $\pm$0.05	**0.30** $\pm$0.06	0.38 $\pm$0.07	11.99 $\pm$0.10	12.03 $\pm$0.16	12.00 $\pm$0.08	12.03 $\pm$0.10
4	(5)	**1.36** $\pm$0.23	1.62 $\pm$0.28	**1.82** $\pm$0.35	2.21 $\pm$0.47	16.27 $\pm$0.09	16.23 $\pm$0.12	16.22 $\pm$0.06	16.26 $\pm$0.15
5	(6)	**3.38** $\pm$0.39	5.17 $\pm$1.65	**4.64** $\pm$0.80	5.16 $\pm$0.64	18.51 $\pm$0.21	18.54 $\pm$0.14	18.53 $\pm$0.12	18.44 $\pm$0.07
6	(7)	**14.00** $\pm$2.83	17.67 $\pm$3.43	**18.07** $\pm$2.13	22.28 $\pm$4.48	22.82 $\pm$0.08	22.78 $\pm$0.10	22.80 $\pm$0.19	22.73 $\pm$0.15
7	(8)	**34.11** $\pm$7.52	38.35 $\pm$4.68	**43.88** $\pm$8.90	55.78 $\pm$12.25	24.96 $\pm$0.12	24.99 $\pm$0.13	24.96 $\pm$0.14	24.95 $\pm$0.14
8	(9)	**108.94** $\pm$19.36	133.06 $\pm$24.75	**175.48** $\pm$33.68	186.91 $\pm$24.64	29.17 $\pm$0.10	29.26 $\pm$0.20	29.23 $\pm$0.14	29.08 $\pm$0.17
9	(10)	**264.59** $\pm$28.50	362.16 $\pm$60.65	**392.91** $\pm$69.13	463.30 $\pm$78.81	31.48 $\pm$0.20	31.54 $\pm$0.15	31.47 $\pm$0.14	31.54 $\pm$0.17
10	(11)	**867.91** $\pm$195.44	1028.70 $\pm$172.62	**1256.52** $\pm$277.94	1509.25 $\pm$195.14	35.78 $\pm$0.09	35.73 $\pm$0.15	35.70 $\pm$0.15	35.84 $\pm$0.16
11	(12)	**2037.65** $\pm$326.63	2454.41 $\pm$377.38	**2941.24** $\pm$457.95	3461.67 $\pm$595.74	38.07 $\pm$0.13	37.98 $\pm$0.15	38.03 $\pm$0.18	38.10 $\pm$0.12

Table 6: Our test setup for the Gripper(b) domain (lower values better). ID refers to the problem ID in the test set. θ refers to the parameters passed to the problem generator for generating the problem. Times indicate the seconds required to find a policy. Similarly, costs are reported as average costs obtained by executing the computed policy for 100 trials. We ran our experiments using a different random seed for 10 different runs and report average metrics up to one standard deviation. Better metrics are at least 5% better and are indicated using bold font.