Learning Generalized Partial Policies from Examples

by

Deepak Kala Vasudevan

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 2020

ABSTRACT

Many real-world planning problems can be modeled as Markov Decision Processes (MDPs) which provide a framework for handling uncertainty in outcomes of action executions. A solution to such a planning problem is a policy that handles possible contingencies that could arise during execution. MDP solvers typically construct policies for a problem instance without re-using information from previously solved instances. Research in generalized planning has demonstrated the utility of constructing algorithm-like plans that reuse such information. However, using such techniques in an MDP setting has not been adequately explored.

This thesis presents a novel approach for learning generalized partial policies that can be used to solve problems with different object names and/or object quantities using very few example policies for learning. This approach uses abstraction for state representation, which allows the identification of patterns in solutions such as loops that are agnostic to problem-specific properties. This thesis also presents some theoretical results related to the uniqueness and succinctness of the policies computed using such a representation. The presented algorithm can be used as fast, yet greedy and incomplete method for policy computation while falling back to a complete policy search algorithm when needed. Extensive empirical evaluation on discrete MDP benchmarks shows that this approach generalizes effectively and is often able to solve problems much faster than existing state-of-art discrete MDP solvers. Finally, the practical applicability of this approach is demonstrated by incorporating it in an anytime stochastic task and motion planning framework to successfully construct free-standing tower structures using Keva planks.

DEDICATION

To my mother, father, and brother for their unconditional love and support.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Dr. Siddharth Srivastava for the continuous guidance, encouragement, and support he has provided me during my time working with him. I also thank the members of the Autonomous Agents and Intelligent Robotics lab for collaborating with me and providing the support I needed, particularly Rushang Karia, for the valuable feedback he has provided me during the course of the writing of this thesis.

Finally, I am extremely grateful to my mother, father, brother, and friends for their unwavering support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

It is well-known in the AI community that Automated Planning is computationally hard. For propositional STRIPS (Fikes and Nilsson (1971)) planning, determining if a given planning problem has a solution is a PSPACE-complete problem (Bylander (1994)). Classical planning is a well-studied area in AI. The solution to a classical planning problem is a sequence of action sequences called a plan. A fundamental assumption classical planning makes is that actions have deterministic effects. This makes their application quite limited in the real world, where many problems have inherent non-determinism.

A non-deterministic environment is one where there is uncertainty in outcomes of action executions. A stochastic environment is a non-deterministic environment where the action outcomes are probabilistic in nature, i.e., each outcome can occur with a fixed probability upon action execution. Markov Decision Processes (MDPs) can be used to model such stochastic environments. Even if the environment is fully observable, plans made for problems in such environments require handling all possible contingencies that might arise during execution. An agent cannot simply use a sequence of actions to reach a goal. Solutions to MDPs that model such environments are used instead. These solutions do not comprise of a sequence of actions, but are policies, which are functions that map all states to actions. For each state the agent currently inhabits, the policy gives the agent the next best action it should execute to reach a goal state with minimum expected cost. For this thesis, a subclass of MDPs which are goal-oriented, discrete, un-discounted and have indefinite horizons are considered.

In many cases, the initial state of an agent is known and computing partial policies that map states in a subset of the state space to actions is sufficient to achieve a goal. There exist many off-the-shelf MDP solvers like LAO* (Hansen and Zilberstein (2001)) and LRTDP (Bonet and Geffner (2003)) that employ heuristic search dynamic programming algorithms to find partial policies, that only prescribe actions to states in a subset of the state space. Although such approaches provide a significant reduction in search time compared to other MDP solvers, they do not always scale well for problems where the number of objects involved is large.

An analysis of partial policies found using MDP solvers for the different problems in the stochastic version of the Keva domain (Kumar (2019)) described in (Shah *et al.* (2020)) showed that common structures were present in solutions across different problems with a varying object counts/names. The state-of-the-art MDP solvers used in (Shah *et al.* (2020)) could not find solutions to problems with large object counts under the given resource constraints. (Shah *et al.* (2020)) use iterative calls to MDP solvers to solve smaller problems and stitch them together to form a policy to perform integrated task and motion planning for larger structures. A key limitation shared by these state-of-the-art solvers is that they do not exploit these common structures observed across different problems to solve each problem. This limitation is shared with approaches to compute classical plans, as described earlier.

Planning Domain Definition Language (PDDL) is an AI planning language first introduced by (McDermott *et al.* (1998)) that uses a planning representation language like STRIPS. It is used to represent classical planning problems in AI. PDDL representations only permit deterministic actions. An extension to PDDL, called Probabilistic PDDL (PPDDL), was developed by (Younes and Littman (2004)) to allow for the representation of domains that consisted of actions with probabilistic effects. This thesis uses PPDDL language to model MDP.

Several advances have been made in classical planning to reduce the computational effort expended in solving a problem over the years. However, despite these advances, many approaches are not practical for solving problems as the number of objects increases. One area of research that tries to address this problem is 'generalized planning'. A generalized plan is an algorithm-like plan that can be used to solve multiple instances of planning problems. Some approaches to generalized planning use abstraction to construct generalized plans from examples of solved instances of planning problems(Srivastava *et al.* (2008),Srivastava *et al.* (2010),Winner and Veloso (2003),Winner and Veloso (2007)). Such approaches try to capture common structures in solved example plans and use them to solve a set of planning problems.

This thesis employs abstraction concepts from generalized planning to recognize and capture repeating patterns observed, for example partial policies. A framework to learn a generalized partial policy from examples of partial policies is developed. The generalized partial policy is represented as a graph. It is shown that this framework finds a unique generalized partial policy graph regardless of the order in which the examples were used.

Even though off-the-shelf solvers for automated planning offer complete solutions to problem instances, there is evidence that an incomplete search routine can yield massive dividends in search time if it can solve such problems quickly (Hoffmann (2000),Hoffmann and Nebel (2001)). The Generalized Policy Instantiation (GPI) method developed as part of this thesis is such an incomplete routine. It instantiates the learned generalized partial policy to find concrete partial policies for previously unseen problem instances quickly yet greedily.

Empirical evaluations were conducted to evaluate GPI's performance against state-of-the-art discrete MDP solvers on several discrete MDP benchmark domains, namely Keva, Delicate-Can-World, Can-World, Hanoi, Gripper, File-World, Rover, and Schedule. An empirical analysis of the results shows that GPI can solve problems much faster than benchmarks solvers. In many cases, the benchmarks solvers failed to find solutions for problem instances containing a large number of objects in them. GPI, however, can find solutions for even such large problems quickly.

An anytime integrated task and motion planning framework developed by (Shah *et al.* (2020)) is used to find solutions for a stochastic task and motion planning problem to build a free-standing Keva tower structure with 12 levels. A state-of-the-art MDP solver used for performing the task planning portion of the framework is replaced with GPI. The practical applicability of instantiating a pre-computed generalized policy to find solutions is showcased in a simulation of the real-world problem using a model of the Yumi IRB-14000 robot as the agent. The reduction in overall time taken to compute task and motion policies for stochastic environments using this approach as compared to when state-of-the-art solvers are used is made apparent.

This thesis's main objectives are to learn a generalized partial policy graph by observing example policies solved by MDP solvers for smaller problems, to capture repeating patterns in example policies such as cycles, in the generalized policy graph. Another objective is to develop a method that can find valid solutions to unseen problem instances by instantiating the generalized partial policy graph. The main contributions of this thesis are to develop:

(1) A framework to learn a generalized partial policy using example policies.

(2) A quick method that instantiates the generalized partial policy to find solutions to new problem instances by directing exploration in the space of policies encompassed by the generalized policy.

Chapter 2

RELATED WORK

In this chapter, related work in forming generalized plans and methods to form contingent plans are explored. Frameworks developed in some related work are utilized in this thesis, particularly that of canonical abstraction. Concepts such as example-based generalized plan formation are very similar to the approach presented in this work to form generalized partial policies. These shall be described in greater detail in this chapter.

(Winner and Veloso (2003)) learn domain-specific planners (dsPlanners) using example partial-order plans. Their approach converts each new plan into a dsPlanner, a combination of `if-else` statements, `while` loops, logical structures, predicates, indicators, and certain operators, by using the DISTILL algorithm. dsPlanners are learned using various example plans and merged with a pre-existing dsPlanner. The domain-specific plans formed are non-looping in nature. Extending upon previous work, (Winner and Veloso (2007)) construct looping dsPlanners, which extend the size of solvable problems. The dsPlanners capture the repeated structures observed in the example plans by identifying matching sub-plans in the example plans, identify unrolled loops in them and convert them into loops. These loops can be causally independent or dependent. A key limitation of these approaches is that they require an initial dsPlanner to be hand-coded. Their approach generalizes only for problems from classical planning that represent deterministic environments.

(Srivastava *et al.* (2008)) presents an approach to learn generalized plans from example plans using canonical abstraction. They classify some unary predicates as abstraction predicates that are used to define the role of an object. The canonical abstraction technique merges different objects of the same role into a summary object. The imprecision that arises from merging objects is modeled using three-valued logic, which assigns a truth value of 0, 1/2, or 1, where 1/2 means 'unknown'. As a result of this role-based abstraction, multiple concrete states can be encapsulated by a single abstract state. Construction of a finite abstract state space, even if the concrete state space is infinite, becomes possible. They generalize example plans by abstracting the concrete states and actions in the plan and then tracing this abstracted example plan to look for recurring abstract state and action sequences, which represent loops. They also provide a classification of problem instances where their generalized plan is guaranteed to work. Their planner "Aranda," produces correct plans for "extended-LL domains". This work also is limited to deterministic settings, in contrast to work presented in this thesis which is applicable in stochastic settings as well.

(Srivastava *et al.* (2010)) showcases a method to form generalized contingent plans using canonical abstraction. This works for situations where there is an uncertainty in object quantities or state properties, but there is a lack of probabilistic information about these uncertainties. This work is more general than the approach presented in this thesis, where the probablities associated with uncertainties involved are known. They use the Three-Values-Logic Analyser (TVLA) framework's functions focus and coerce to draw-out a representative element from summary elements. The focus operator on the 'chosen' formula produces a set of abstract structures, and the coerce operator determines which of these structures is valid. Structures formed which are not consistent with the example policy π at the same step represent possible situations that are not handled by π. These can be marked as loose ends, using which they generate additional concrete plans to handle these situations. The Branch and Merge algorithm addresses combining multiple example plans by accurately determining segments of the existing plans where the new plans would be useful. Their approach is more general when compared to the work presented in this thesis. This thesis focuses on stochastic problems, and uses off-the-shelf MDP solvers to find contingent trees derived from optimal policies. Hence, the branches appear in examples itself, and only merging is done while forming the generalized policy, making the problem of constructing generalized policies easier.

(Srivastava *et al.* (2011a)) present an algorithm for generalizing example plans called "ARANDA-Learn". They use canonical abstraction to form a graph-based generalized plan which can contain simple loops. This algorithm first finds a 'trace' of the example plan, an abstracted SAS sequence. Then, identification of loops in this SAS sequence is done. A graph containing these identified loops is formed. This graph represents the generalized example plan. They show that one iteration of any loop formed by their approach is guaranteed to make measurable progress towards the goal. Their approach also efficiently computes plan preconditions under which for a subclass of problems, the loop of actions formed is guaranteed to terminate and lead to the desired goal state. Similar to previous work, they construct generalized plans by using a single example concrete plan. Thus, the domain coverage of generalized plans formed may be limited when compared to constructing plans using the approach presented in this work.

(Srivastava *et al.* (2011b)) propose an algorithm for generalized plan synthesis. The Hybrid Generalized Plan Synthesis algorithm incrementally improves the generalized plan by identifying problem instances it cannot solve. It then invokes a classical planner to solve this instance, computes the solution's trace, and merges it back to the generalized plan. They identify unsolved problem instances using open nodes, which are terminal non-goal nodes. The algorithm runs until all open nodes are resolved or the resource limit is reached. The approach presented in this thesis does not automatically generate new example plans that can be used to resolve unhandled situations in the generalized policy formed, but can determine if a new example policy is useful or not. This work also performs tracing to form branches in the generalized plan, unlike the work presented in this thesis where the branches are already present in the concrete policy.

The following five papers discuss different approaches to generalized planning.

(Bonet *et al.* (2009)) present a model-based method to derive finite-state controllers automatically, which can be used to find solutions for a class of contingent problems with deterministic actions and partial observability. They convert the task of deriving a controller for models into a conformant planning problem that is solved using a classical planner using a sound and complete method to transform confromant plans to classical plans (Palacios and Geffner (2007)). The derived controllers are generalized, where apart from the original problem, they can also solve problems of different sizes or different uncertainty of initial situation / action effects. Similar to their approach, the approach presented in this thesis also does not require sensing actions. However, their approach works only for a limited set of problems, where actions have no preconditions.

(Belle and Levesque (2016)) explore the applicability and correctness of generalized plans in domains that are possibly unbounded, and/or stochastic/continuous. They introduce a generic controller framework to capture different type of planning domains, and present various notions of adequacy based on termination, boundedness of plan length, cyclicity of plans formed and goal satisfaction criteria. They present theoretical results relating the different notions of adequacy when using their controller framework. They also present probabilistic analogues to the adequacy notions presented for both discrete and continuous settings, and present theoretical findings for the same. Finally, they show how the adequacy notions developed relate to the problems of goal reachability and goal achievability. The stochastic domains we consider are stochastic in nature. Their theoretical results pertaining to discrete probabilistic adequacy are applicable to domains considered in this thesis if generalized plans were constructed using their controller framework. Although this thesis does not produce any such theoretical results, this work provides an interesting direction to explore termination and goal-reachability guarantees.

(Sanner and Boutilier (2009)) present a technique to translate a subset of PPDDL problems into a first-order MDP (FOMDP) (Boutilier *et al.* (2001)). Their approach finds a solution directly at the relational level as opposed to traditional approaches which apply to ground factored representation of MDPs, which have high time and space complexity. Their approach can derive a domain-independent policy without grounding at any intermediate step. Using first-order algebraic decision diagrams (FOADDs), they exploit logical structures in problems to find FOMDP solutions. (Sanner and Boutilier (2009)) generalize linear programming techniques for MDPs to FOMDPs, and showcase the performance of their first-order approximate linear programming (FOALP) planner. Although not optimal, their approach provides solutions that are close to optimal in many cases. Unlike the approach presented in this work, they do not use examples to construct generalized policies.

(Segovia-Aguas *et al.* (2019)) represent generalized plans as planning programs using structured programming mechanisms of control flow and procedure calls, which allow for compact representation, and representation of hierarchical and recursive solutions respectively. They also present an approach to ”compile” generalized plans into classical plans, which allows them to compute generalized plans using an off-the-shelf planner. They also present approaches to compute planning programs for non-deterministic settings, and also show how the compilation they perform can be extended to compute high-level features. Simple classification tasks from machine learning can be computed using planning programs they develop with this extension.

Their approach is applicable only for deterministic settings. Also, the overall control flow present in programs they compute is captured by the generalized policy graph constructed in this thesis in the form of abstract state-action transitions.

(Segovia-Aguas *et al.* (2020)) extend the notion of validation for generalized plans as the problem of verifying if a given generalized plan can solve a set of input *positive* instances while failing to do so on a set of *negative* instances. This notion allows for using quantitative metrics used in machine learning to assess generalization capacity of generalized plans. They incorporate this notion of plan validation into a compilation for plan synthesis (Segovia-Aguas *et al.* (2019)) that takes both positive and negative instances as inputs. Experiments they conducted indicate that using negative examples accelerates generalized plan synthesis. Their approach differs from the one presented in this thesis in that it is limited to deterministic settings. The approach presented in this thesis does not utilize negative examples in the construction of generalized policies and is thus limited in this front.

(Peot and Smith (1992)) explore conditional non-linear plans (CNLP) that contain contingent branches that account for predefined sources of uncertainty. They use three-valued logic to represent truth values of propositions in a state (true, false, and unknown). They define conditional actions that can have mutually exclusive effects where a proposition whose value is unknown is resolved into either true/false and labeled with a unique observation label α / $\neg\alpha$. A step is an instance of an action appearing in a plan. Any step that occurs at any point after a step involving a conditional action inherits the unique observation label of that conditional action in its *context*. A *reason* of a step is the set of goals that are reachable from that step. CNLP first constructs a non-contingent plan. If a plan is found such that the goal is reached for all possible observations, the planner stops. Otherwise, CNLP observes a set of observation labels which do not lead to the goal and attempts to construct a new plan branch. A key limitation of this approach is that CNLP does not generalize to new examples. It has to solve each instance of the problem from scratch.

Solving real-world problems in robotics require agents to compute high-level abstract plans, along with low-level motion planning in the physical world. In stochastic scenarios, contingent plans are required to complete a task. The plans in these cases take the form of policies. In many cases, these high-level policies alone are insufficient to solve the problem since a lot of information about the physical environment is lost due to abstraction. (Shah *et al.* (2020)) present an approach to perform integrated task and motion planning in stochastic scenarios. The anytime feature is showcased, which reduces the probability of encountering an unresolved contingency over time. They prove the probabilistic completeness of their algorithm. Their work requires the use of an MDP solver, which can be substituted with a solver that uses any generalized planning framework, thus providing a reliable framework to demonstrate real-world applicability.

Chapter 3

BACKGROUND

3.1 Markov Decision Processes

The domains considered in this thesis focuses on three sub-classes of MDPs, all of which have a finite state space and action set and have a stationary transition and action cost function.

Stochastic shortest path MDPs (SSPs) (Bertsekas and Tsitsiklis (1991)), are MDPs that assume that there exists at least one complete proper policy, which reaches a goal from any state in the state space with a probability 1. They also assume that every improper policy incurs infinite cost. A proper policy for SSPs is one that is guaranteed to reach a goal from any state in the policy. An improper policy contains states from which goals are unreachable. An optimal policy for an SSP MDP will take a finite time to reach a goal.

An SSP MDP is defined as an 8 tuple $\Gamma = \langle S, A, T, C, \mathcal{G}, s_{init}, P, O \rangle$ where S is a finite state space, A is a finite set of stochastic actions, $T : S \times A \times S \rightarrow [0, 1]$ is a stationary transition function, $C : S \times A \times S \rightarrow \mathbb{R}$ is an action cost function, $\mathcal{G}$ is a goal condition, $s_{init} \in S$ is the initial state, P is a set of predicates and O is a set of objects.

The set of predicates with arity k is denoted as P^k. An atom $p^k(\mathfrak{o})$ is defined as a predicate $p^k \in P^k$ instantiated with an object list $\mathfrak{o} \in O^k$, where $\mathfrak{o} = \langle o_1, \ldots, o_k \rangle$ $o_1, \ldots, o_k \in O$. A literal is an atom with a truth value assigned to it (true/false). A state is defined as a conjunction of literals. The goal condition $\mathcal{G}$ is a conjunction of literals that describe a set of states which represent goal states in the MDP. A stochastic action $a \in A$ is defined by its preconditions, effect list, and parameters. The precondition of an action *pre* is a conjunction of literals that must be true for the action to be applicable in a state s i.e. $pre \subseteq s$. The effect list is defined as $\epsilon_a = \langle(\mathfrak{p}_1, \mathfrak{e}_1), \ldots, (\mathfrak{p}_j, \mathfrak{e}_j), \ldots, (\mathfrak{p}_k, \mathfrak{e}_k)\rangle$, where $\mathfrak{p}_j \in [0,1]$ represents the probability of the action resulting in effect $\mathfrak{e}_j$ (Younes and Littman (2004)). Each effect $\mathfrak{e}_j$ is a conjunction of positive($\mathfrak{e}_j^+$) and negative($\mathfrak{e}_j^-$) literals that describe how the state s changes when the action is executed. The action described above has $|\epsilon_a|$ effects. It is required that $\sum_{j=1}^{k} \mathfrak{p}_j = 1$. The code developed to implement this thesis does not consider conditional effects, or disjunctive preconditions. An action is instantiated with an object list as its parameters as $a(\mathfrak{o})$ where $\mathfrak{o} = \langle o_1, \ldots, o_n\rangle$, $o_1, \ldots, o_n \in O, \mathfrak{o} \in O^n$. This action is said to have n parameters. Unless specified, stochastic actions will be referred to as actions. The transition function T maps states and actions to successor states and assigns such a transition with probabilities. $\forall s, s' \in S$ where $s' = (s \setminus \mathfrak{e}_j^-) \cup \mathfrak{e}_j^+$, $a \in A$, $T(s, a, s') = \mathit{Probability}(s'|s, a) = \mathfrak{p}_j$.

SSP's make assumptions that prevent them from being used to model catastrophic situations where a state has no path to a goal. Such states are called *dead-end* states. For such a state $s_{dead} \in S$, no state $s \in S, s \models \mathcal{G}$ is reachable. (Kolobov *et al.* (2012)) describe two extensions to SSPs that help model such situations. They are SSPs with avoidable dead-ends (SSPADE) and SSPs with unavoidable dead-ends with finite penalty (fSSPUDE).

SSPADE (Kolobov *et al.* (2012)) relax the requirements posed by SSPs. Such problems only require an initial state to be known, and at least one proper policy rooted at the initial state exists. Here, dead-ends are avoidable from the initial state.

In fSSPUDEs (Kolobov *et al.* (2012)), the initial state is required to be known as well. However, dead-end states are unavoidable from the initial state in all possible policies. No proper partial policies exist for such problems. Whenever an agent reaches a state where the expected cost is greater than a predefined finite value, the agent simply pays a finite penalty equal to this value and gives up. (Kolobov *et al.* (2012)) show that all fSSPUDE problems can be represented as SSP problems.

The solution Γ is a policy $\pi : S \rightarrow A$. Since the initial state is known, a partial policy rooted at s_{init}, that maps only a subset of the state-space containing states reachable from s_{init} to the action set is sufficient to reach a goal. A partial policy is defined as $\pi_{s_{init}} : S' \rightarrow A, S' \subseteq S$, all $s' \in S'$ are reachable from s_{init}. Unless specified, partial policies will be referred to as policies. The optimal solution to the problem Γ is a policy π^* that reaches the goal with the least expected cumulative cost.

Every policy $\pi_{s_{init}}$ can be represented by a policy graph $G = \langle V_G, E_G \rangle$ where V_G and E_G are the vertices and edges of G respectively. A labelling function $\mathcal{L}_G : V_G \rightarrow S \times A$ that maps each vertex $v_i \in V_G$ to a *state-action pair* $\langle s_i, a_i \rangle$, where $s_i \in S$ represents the current state and $a_i \in A$ represents an action given by $\pi_{s_{init}}(s_i)$. There exists a root vertex $v_{root} \in V_G$, for which $\mathcal{L}(v_{root}) = \langle s_{init}, \pi(s_{init}) \rangle$. Each node v_i has $|\epsilon_{a_i}|$ child nodes. Each child node contains states that are a result of applying action a_i on state s_i with one of its non-deterministic effects from ϵ_{a_i} observed. The transition probabilities from one node to the next is determined by T. Every edge $e_{ij} \in E$ is an ordered pair $e_{ij} = \langle v_i, v_j \rangle, v_i, v_j \in V_G$ forming a directed link from vertex v_i to v_j in G and represents a *state-action transition.* The set of all possible concrete state-action pairs is $S \times A$, and the set of all possible state-action transitions is $S \times A \times S \times A$.

A goal node is a leaf node in G, where $v_{goal} \in V_G, \mathcal{L}_G(v_{goal}) = \langle s_{goal}, a_{goal} \rangle$, where $s_{goal} \in S, s_{goal} \models g$ and $a_{goal} \in A$ is a dummy action that contains no parameters, preconditions or effects.

A dead end node is a leaf node in G, where $v_{dead} \in V_G, \mathcal{L}_G(v_{dead}) = \langle s_{dead}, a_{dead} \rangle$, where $s_{dead} \in S, \neg s_{dead} \models g$ is a dead-end state , and a_{dead} is a dummy action that contains no parameters, preconditions or effects.

MDPs find use in many real-world problems. One such instance is in anytime integrated task and motion planning for stochastic environments, where a robot in the real world has to act in a stochastic environment in order to achieve a goal. This is described in the next section.

3.2 Anytime Integrated Task And Motion Policies For Stochastic Environments

A framework to compute anytime integrated task and motion policies for stochastic environments was developed by (Shah *et al.* (2020)). They define a stochastic task and motion planning problem (STAMPP) as a tuple $\langle \mathcal{M}, c_0, \alpha, [\mathcal{M}] \rangle$, where $\mathcal{M}$ is the high-level policy, c_0 is the initial concrete configuration of the environment, α is a composition of function and entity abstractions, and $[\mathcal{M}]$ is the abstraction of $\mathcal{M}$ obtained using α. They present the ATM-MDP Algorithm, which accepts $[\mathcal{M}]$, a domain $\mathcal{D}$, problem $\mathcal{P}$, and SSP MDP solver *SSP* and a motion planner M. Overall, the ATM-MDP algorithm builds a plan refinement graph (PRG), and interleaves computation among the two processes: *(1) Concretization of an abstract policy, (2) Updating abstraction for a fixed concretization and computation of a new abstract policy.* For the first process, feasible refinement of a partial path in PRG is searched by instantiating its symbolic action arguments with values from their original non-symbolic domains. An example of this is where abstract symbols representing trajectories are refined using motion planners to calculate physical low-level motion plans until no unrefined paths are left, or some resource limit has reached. Sometimes, there is no feasible refinement available. In such cases, the second process is used. Here, (Shah *et al.* (2020)) fix a concretization for the partially refined path and update the earliest abstract state in this path whose subsequent concretizations were infeasible. The subsequent portion of the policy is discarded, and the failed preconditions learned from the environment are updated in this abstract state. An updated state is formed using this, and the SSP solver is used to compute a new policy from this state.

For long-horizon tasks, computation of refinements for multiple contingencies with a low probability of being encountered may be wasteful computations to perform in real-time settings. An optimization the authors of (Shah *et al.* (2020)) developed to avoid such computation is summarized in their anytime-approach. Here, they use a greedy approach, where they prioritize the selection of paths to refine based on the probability of encountering that path $\mathfrak{p}$, and the cost of refining that path c. $\mathfrak{p}/c$ is computed for all paths, and the ones with the highest value are refined first. They prove that if a proper policy that reaches a goal state within a predefined horizon exists, then their approach will find it with probability 1.0. After extensive domain investigation, a preliminary version of the approach developed in this thesis was used to stitched together solutions for multiple smaller problems to calculate the solution for a large problem. This thesis implements a framework using canonical abstraction to automatically learn patterns from solutions to smaller problems and use them to solve larger problems.

3.3 Canonical Abstraction

Canonical abstraction was originally developed as a state abstraction technique from software model checking Sagiv *et al.* (2002). It helps in representing an unbounded set of concrete states compactly.

For finding the canonical abstraction of states in a domain, a subset of unary predicates $P_{abs} \subseteq P^1$ are identified as *abstraction predicates*. In this thesis, all unary predicates in a domain are considered as abstraction predicates i.e. $P_{abs} = P^1$.

Definition 1. *(Role) The role of an object $o \in O$ in a state $s \in S$ is the set of abstraction predicates that it satisfies: $role(o) = \{p_{abs} | p_{abs} \in P_{abs}, p_{abs} \in s\}$.*

The maximum possible number of roles in any state from problems derived from a domain with $|P_{abs}|$ abstraction predicates is $2^{|P_{abs}|}$.

$\psi(r) = \{o | o \in O, role(o) = r\}$ defines the set of objects associated with a particular role r. The following example introduces canonical abstraction with the help of a problem instance from the Keva domain.

Example 1. *A Keva problem consists of two locations and a single gripper used to pick multiple planks and place them on a table to build some structure. Consider the following state in the Keva problem with objects $O = \{l_0, l_1, p_0, p_1\}$, where l_i represents the the i^{th} location and p_j represents the j^{th} plank. $s = \{free^1(l_0), free^1(l_1), ingripper^1(p_0), ontable^1(p_1), placed^1(p_1), clear^1(p_1)\}$. The roles for objects in state s are $role(p_0){=}r_0, role(p_1){=}r_1, role(l_1){=}role(l_2){=}r_2$; $r_0{=}\{ingripper^1\}, r_2{=}\{free^1\}$, $r_1{=}\{clear^1, ontable^1, placed^1\}$. So, $\psi(r_0){=}\{p_0\}$, $\psi(r_1){=}\{p_1\}$ and $\psi(r_2){=}\{l_0, l_1\}$.*

An object list $\mathfrak{o} = \langle o_1, \ldots, o_k \rangle$, $o_1, \ldots, o_k \in O$, can be used to compute a role list $\ddot{\mathsf{r}} = \langle role(o_1), \ldots, role(o_k) \rangle$ for a given state s. $\Psi(\ddot{\mathsf{r}}) = \psi(role(o_1)) \times \ldots \times \psi(role(o_k))$ defines the set of object lists whose corresponding role list is $\ddot{\mathsf{r}}$.

The canonical abstraction of a state is found by converting each literal in that state $p^k(\mathfrak{o}) \in s$ to an abstract literal $\bar{p}^k(\ddot{\mathsf{r}})$, $\bar{p}^k \equiv p^k$, where $\ddot{\mathsf{r}}$ is the corresponding role list for object list $\mathfrak{o}$, and then grouping common abstract literals. The imprecision in truth values resulting from this grouping is modeled using three-valued logic.

Definition 2. *(Canonical Abstraction) Canonical abstraction of a concrete state $s = \{p^k(\mathfrak{o}) \mid p^k \in P, \mathfrak{o} \in O^k\}$ is an abstract state $\bar{s} = \{\bar{p}^k(\ddot{\mathsf{r}}) \mid \bar{p}^k \equiv p^k\}$. The truth value of $\bar{p}^k(\ddot{\mathsf{r}})$ in $\bar{s}$ is given by:*

- $\bar{p}^k(\ddot{\mathsf{r}}) = 0 \iff \forall(\mathfrak{o} \in \Psi(\ddot{\mathsf{r}}))\ p^k(\mathfrak{o}) \notin s$
- $\bar{p}^k(\ddot{\mathsf{r}}) = 1 \iff \forall(\mathfrak{o} \in \Psi(\ddot{\mathsf{r}}))\ p^k(\mathfrak{o}) \in s$
- $\bar{p}^k(\ddot{\mathsf{r}}) = 1/2 \iff (\exists(\mathfrak{o} \in \Psi(\ddot{\mathsf{r}}))\ p^k(\mathfrak{o}) \in s) \bigwedge (\exists(\mathfrak{o} \in \Psi(\ddot{\mathsf{r}}))\ p^k(\mathfrak{o}) \notin s)$

Example 2. *The canonical abstraction of state s described in Example 1 is* $\bar{s} = \{free^1(r_2), ontable^1(r_0), placed^1(r_0), clear^1(r_0), ingripper^1(r_1)\}$. *The truth values are* $free^1(r_2){=}1, ingripper^1(r_1){=}1, ontable^1(r_0){=}1, placed^1(r_0){=}1, clear^1(r_0){=}1$.

Consider another state from Keva domain with an extra plank on the table compared to the one shown in Example 1. $s' = \{free^1(l_0), free^1(l_1), ontable^1(p_0),$ $placed^1(p_0), clear^1(p_0), ontable^1(p_1), placed^1(p_1), clear^1(p_1), ingripper^1(p_2)\}$.
The canonical abstraction of this state is also given by $\bar{s}$. The abstract state $\bar{s}$ captures all states in Keva domain, where one plank is present in the gripper and all other planks are placed on the table. This demonstrates the usefulness of canonical abstraction, where different states with similar properties can be mapped to the same abstract state. It allows for representation of an unbounded numbers of objects and can be used to identify recurring state properties in policies.

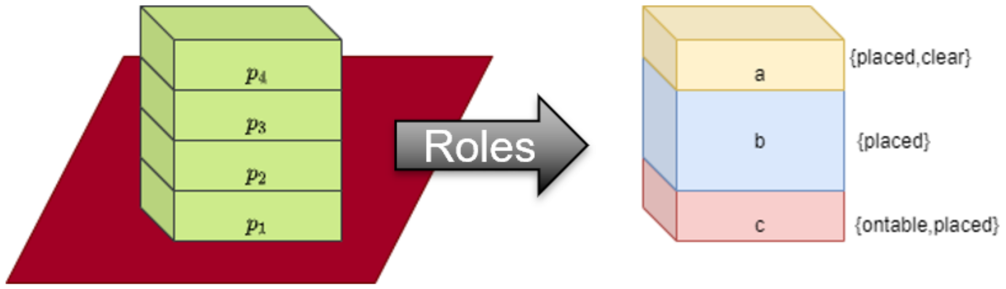


Figure 3.1: Objects And Associated Roles For A Keva Domain State

Consider the state of the system where 4 Keva planks are stacked together as shown in Fig. 3.1. The state of the system is given by:

$s = \{placed^1(p_1), placed^1(p_2), placed^1(p_3), placed^1(p_4), clear^1(p_4), ontable^1(p_1),$ $onsingleplank^2(p_2, p_1), onsingleplank^2(p_3, p_2), onsingleplank^2(p_4, p_3),$

The binary predicate $onsingleplank^2(p_2, p_1)$ represents the relation between p_2 and p_1, where p_2 is on a single plank p_1. Three distinct roles a,b and c are present in this

$onsingleplank^2$	a	b	c
a	0	1/2	0
b	0	1/2	1/2
c	0	0	0

Table 3.1: Truth Value Assignment For Binary Predicates

state as shown in the figure on the right. The truth values for $onsingleplank^2$ in the canonical abstraction of the state is shown in Table 3.1. $onsingleplank^2(a, b) = 1/2$ because although there exist some concrete literals where planks with role 'a' are one those with role 'b', this relationship is not true for all planks with roles 'a' and 'b'. In a similar fashion, the truth values are computed for all permutations of roles.

An n-parameter action $a(\mathfrak{o})$ where $\mathfrak{o}^n \in O^n$ can be abstracted as $\bar{a}(\ddot{\mathsf{r}})$, $\bar{a} \equiv a$. The following example describes action abstraction.

Example 3. *Consider action a=putdown_plank_ondoubleplank$(yumi, p_2, p_1, p_0)$. Let roles associated with each object be r_1=role$(yumi)$=$\{\}$, r_2=role(p_2)=$\{ingripper^1\}$, r_3=role(p_1)=role(p_0)=$\{clear^1, ontable^1, placed^1\}$. The corresponding abstract action is $\bar{a}$ = putdown_plank_ondoubleplank(r_1, r_2, r_3, r_3).*

The set of all abstract states is denoted by $\bar{S}$, and the set of all abstract actions is denoted by $\bar{A}$. The set of all possible abstract state-action pairs is $\bar{S} \times \bar{A}$, and the set of all possible state-action transitions is $\bar{S} \times \bar{A} \times \bar{S} \times \bar{A}$.

Chapter 4

METHODOLOGY

The overall methodology presented in this thesis can be divided into two sections. The first involves learning the generalized policy, and the second describes how the learnt policy can be instantiated to solve new problems. Each of these sections fulfill individual objectives of this thesis that were outlined in the introduction.

4.1 Learning Generalized Policies

Learning the generalized policy for a given stochastic domain can be broadly simplified into three steps: (1) Generating example policies, (2)Tracing, and (3) Merging. This is described in Fig. 4.1.

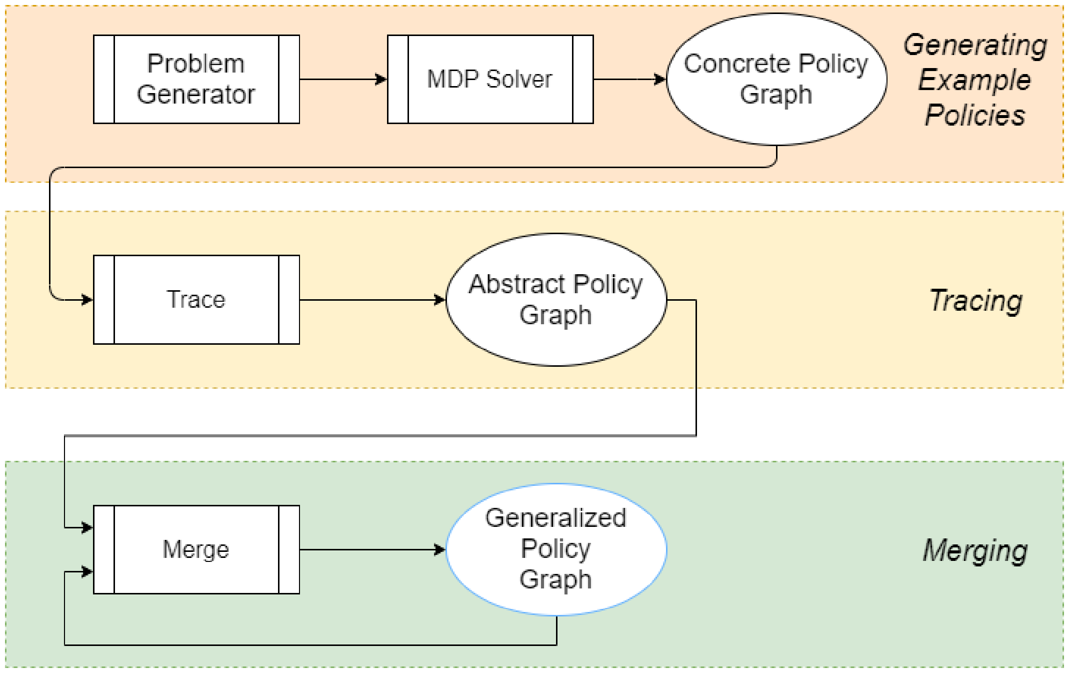


Figure 4.1: Learning Generalized Policies

4.1.1 Generating Example Policies

For each stochastic domain considered in this thesis, a domain file is written in PPDDL. A problem generator is implemented to generate problem instances for each domain. All problem instances contained similar initial states and goal conditions for a particular domain.

(Hansen and Zilberstein (2001)) present LAO*, which is a heuristic search dynamic programming algorithm that can be used to solve MDPs. The solutions for such problems are policies which can be represented as AND-OR graphs. The LAO* algorithm can solve problems that have cyclic solutions. (Hansen and Zilberstein (2001)) show that LAO* can be used to find optimal partial policies rooted at an initial state, which are solutions to a goal-directed MDP problem without expanding the entire state space. An open-source implementation (Pineda (2014)) is used for implementation of LAO*. A problem generator generates a domain file and a problem instance as inputs to an LAO*, which returns an optimal partial policy. Since this is an optimal solution, it will take a finite amount of time to reach the goal. This partial policy is unrolled into a tree G with a bounded depth in a similar fashion as done in (Shah *et al.* (2020)). Domain analysis reveals the appropriate depth required for each problem. G is referred to as the concrete policy graph.

An example concrete policy graph from the Keva Domain is as shown in Fig. 4.2. The values of $\mathcal{L}_G(v)$ are written inside the circle representing each vertex $v \in V_G$. Here $s_1 = \{free^1(l_0), free^1(l_1), handempty^1(yumi)\}$ represents the initial state. a_1 corresponds to $human\_place(yumi, p_1)$, where a human places a plank non-deterministically in either of the two locations l_0/l_1. The resulting successor states are s_2 and s_3. This action is assigned to the initial state by the policy found by LAO*. In a similar fashion, the entire graph is constructed.

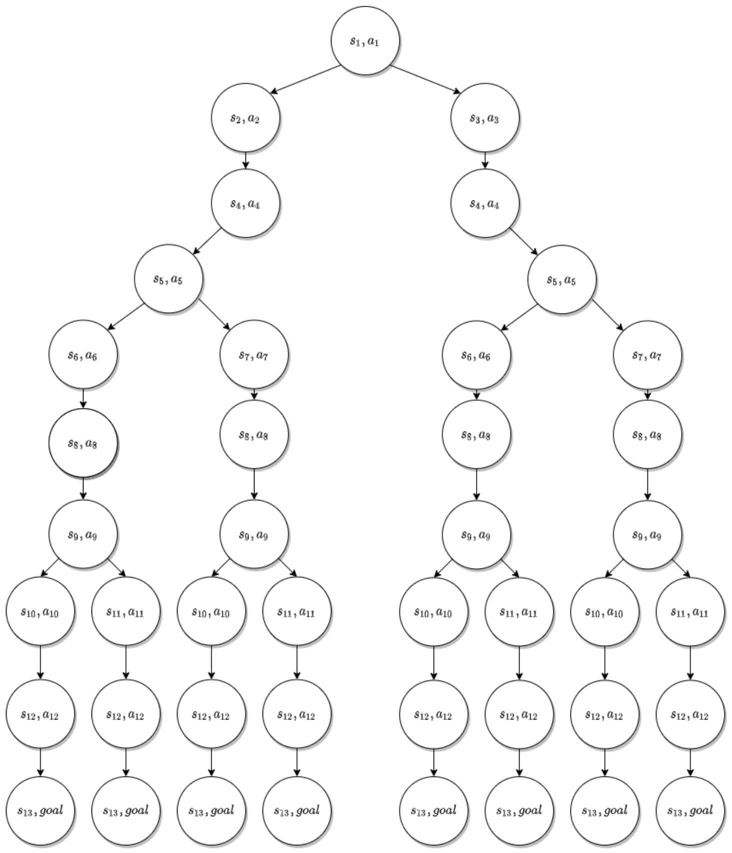


Figure 4.2: Example Concrete Policy

4.1.2 Tracing

The trace of an example policy G is a graph $G' = \langle V_{G'}, E_{G'} \rangle$, and a labeling function $\mathcal{L}_{G'}$, which helps in identifying the underlying abstract structure present in an example policy. The algorithm for trace is shown in Algorithm 1.

```
Input  : $G, \Gamma$
Output: $G'$
1 $G' = AbstractConcPolicy(G, \Gamma)$
2 $R_{reps} = IdentifySubtreeRepeats(G')$
3 $G' = DeleteRedundantRepeats(G', R_{reps})$
4 $G' = FindDeadEnds(G', G, \Gamma)$
5 return $G'$
```

Algorithm 1: Trace

Converting an example policy to a trace is summarized in 4 steps as described in Section 4.1.2.1 to Section 4.1.2.4.

4.1.2.1 Abstracting Concrete Policy

This section discusses the function *AbstractConcPolicy* from Line 1 of Trace Algorithm.

An abstraction function $\Upsilon : S \times A \rightarrow \bar{S} \times \bar{A}$ is a function that maps set of all concrete state-action pairs to abstract state-action pairs using the canonical abstraction framework. $\Upsilon(\langle s_i, a_i \rangle) = \langle \bar{s}_i, \bar{a}_i \rangle, \langle s_i, a_i \rangle \in S \times A, \langle \bar{s}_i, \bar{a}_i \rangle \in \bar{S} \times \bar{A}$.
Given a concrete policy graph $G = \langle V_G, E_G \rangle$ with a labeling function $\mathcal{L}_G$, an abstract policy graph is defined as $G' = \langle V_{G'}, E_{G'} \rangle$ such that G and G' are isomorphic graphs i.e. $G \cong G'$, with some isomorphism function $f : V_G \rightarrow V_{G'}$. A labeling function for $\bar{\mathcal{L}}_{G'}$ maps each vertex in $V_{G'}$ to an abstract state-action pair. $\bar{\mathcal{L}}_{G'} : V_{G'} \rightarrow \bar{S} \times \bar{A}$. For all $v_{abs} \in V_{G'}, v_{conc} \in V_G$, and $v_{abs} = f(v_{conc})$, $\bar{\mathcal{L}}_{G'}(v_{abs}) = \Upsilon(\mathcal{L}_G(v_{conc}))$.

Since both S, and A are finite, $\bar{S}$ and $\bar{A}$ are finite as well. Hence, $\bar{S} \times \bar{A}$ is a finite set. So, there exists only a finite set of abstract state-action pairs.

Hence, the set of abstract state action transitions $\bar{S} \times \bar{A} \times \bar{S} \times \bar{A}$ is also finite.

In this manner, all the concrete state-action pairs present in G are used to calculate their equivalent abstract state-action pairs. Using this, the corresponding state-action transitions are found.

The corresponding abstracted policy graph of the example concrete policy graph showin in Fig. 4.2 is shown in Fig 4.3. The values of $\bar{\mathcal{L}}_{G'}(v)$ are written inside the circle representing each vertex $v \in V_{G'}$. Note that in the resulting graph, different concrete states/actions can map to the same abstract state/action.

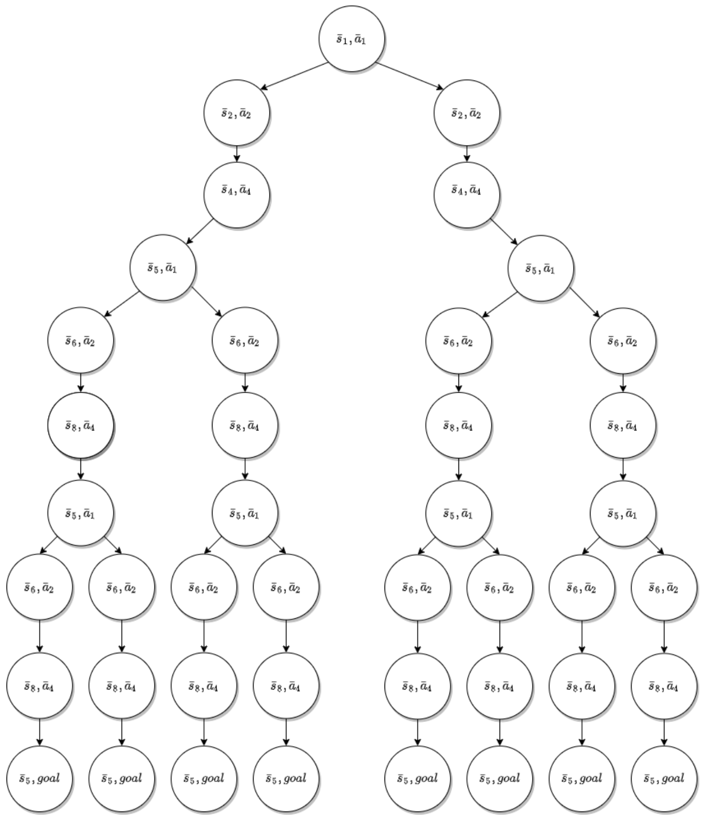


Figure 4.3: Example Abstracted Policy

4.1.2.2 Identifying Repeating Structures

This section discusses the function $IdentifySubtreeRepeats$ seen in Line 2 of Algorithm 1. Repeating structures in the tree G' are sub-trees of G' that are identical to one another. A set of identical sub-trees are called "sub-tree repeats". (Flouri *et al.* (2013)) present a method to identify all non-overlapping sub-tree repeats in unrooted, unordered, labeled trees. Non-overlapping sub-tree repeats shall be referred to as repeats for the sake of simplicity. The example policies obtained from the previous step are rooted, unordered, and labeled trees. A modified version of the FORWARD-STAGE algorithm shown in (Flouri *et al.* (2013)) is implemented to identify the repeating structures of rooted labeled trees. The node corresponding to the initial state is the root node, and the labels used as input to the algorithms is the set of unique identifiers $\bar{\mathcal{L}}_{G'}$ for each vertex $v \in V_{G'}$. Inputting this to the FORWARD-STAGE algorithm gives us all the repeats present in the example policy. The complexity of the algorithm presented in (Flouri *et al.* (2013)) is linear in the number of nodes in the tree. If G' has depth d, then the output of this algorithm is a dictionary R_{reps} with keys $k \in \{1, 2, ..., d\}$. The corresponding value of $R_{reps}[k]$ is the set of root nodes of repeating sub-trees at depth k, along with their corresponding labels l_i assigned by the FORWARD-STAGE algorithm. Note that depth here refers to the minimum distance of a node to any leaf node, with leaf nodes having a depth of 1. Two nodes with the same label l_i in $R_{reps}[k]$ indicate that they both represent the root nodes of repeating sub-trees.

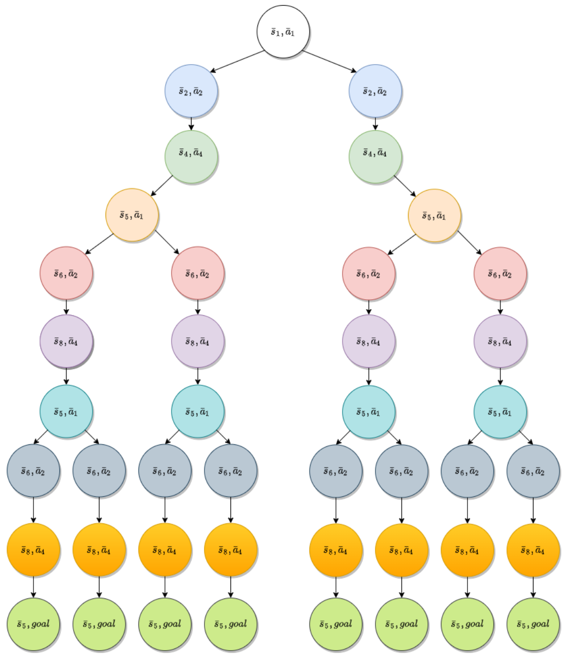


Figure 4.4: Identified Repeating Sub-Trees

Fig 4.4 illustrates the identified sub-tree repeats from the previously calculated abstract graph, by assigning a unique color to each root node of a repeating sub-tree

4.1.2.3 Deleting Repeating Structures

This section discusses the function *DeleteRedundantRepeats* seen in Line 3 of Algorithm 1. Algorithm 2 describes *DeleteRedundantRepeats*, which uses information learned about the repeats in the previous step to remove redundant structures.

```
Input   : $G', R_{reps}$
Output: $G'$ with no repeating structures
$levels = R_{reps}.keys()$
for $i$ in $levels$ do
    $R = R_{reps}[i]$
    $U = GetUniqueLabelRootMap(R, G')$
    for $(root\_node, label)$ in $R$ do
        if $U(label) == root$ then
            continue
        else
            $unique\_root = U(label)$
            $G'.add\_edge(root\_node.parent, unique\_root)$
            $G'.delete\_subtree(root)$
return $G'$
```

Algorithm 2: DeleteRedundantRepeats

For each repeat, one sub-tree is preserved while the rest are deemed redundant. As a redundant sub-tree is deleted, an edge is added from the parent of the root node of the redundant sub-tree to the root node of the preserved sub-tree. The process of deletion is done in a top down approach. This means that redundant sub-trees with higher levels, i.e., lower distance from the root node, are deleted first. The resulting graph generated by this algorithm contains no repeats. The function *GetUniqueLabelRootMap* converts an input list of tuples $R = \{\langle r_1, l_1\rangle, \langle r_2, l_2\rangle, \ldots, \langle r_n, l_n\rangle\}$ into a map $U : \{l_1, \ldots, l_n\} \rightarrow \{r_1, \ldots, r_n\}$. $U(l_i) = r_i$, if for some $l_i \in \{l_1, \ldots, l_n\}, \exists r_i \in \{r_1, \ldots, r_n\}$ s.t. $\langle r_i, l_i\rangle \in R, r_i \in V_{G'}$.

The abstract policy tree after removal of the sub-tree repeats present in it is shown

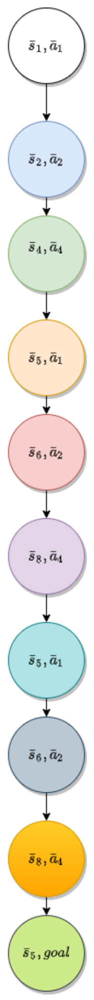


Figure 4.5: Abstracted Policy After Deleting Sub-Tree Repeats

in Fig. 4.5. It can be observed that unique labels are preserved at each depth of the tree while redundancies are deleted. In the Keva example, the resulting graph is a linear sequence of abstract-state action pairs. This is not necessarily true for other domains. In general, the resulting graph formed is a directed acyclic graph.

4.1.2.4 Finding Dead-Ends

The leaf nodes present in G' can be classified into two types: abstract representations of goal nodes, and non-goal nodes. Goal nodes contain states that satisfy the goal condition $\mathcal{G}$. Non-goal nodes contain states which do not satisfy the goal condition, and may be assigned an action by the policy. If the a_{dead} action is the corresponding action in such a node, then it is classified as a dead-end node. If any other action is assigned, it is not clear if the node represents a dead-end or not. Consider such a node called v_{leaf}. For determining this, certain checks are performed. First, check if a state-action pair in the leaf node corresponds to some internal node in G' from which a path to a goal node exists. If such a node exists, then such leaf nodes are ignored as they represent conditions where a path to the goal exists. If no such node exists, then a new example policy is attempted to be found. This new example policy is found by solving a new MDP problem Γ', where Γ' is similar to Γ except that s'_{init} for Γ' is given by the state from $\mathcal{L}_G(f^{-1}(v_{leaf}))$, where f is the isomorphism function used in abstracting the concrete policy. If Γ' is solvable, the trace of the resulting solution for Γ' is calculated and merged with the current trace G' by connecting an edge from v_{leaf} to the root node of the resulting trace. If not, v_{leaf} is marked as a dead-node, with the corresponding action in the state-action pair changed to the dummy action 'dead'.

4.1.3 Merge

Using the merge algorithm, a graph based generalized partial policy is generated using traces. A graph-based generalized partial policy is a graph $\mathcal{H} = \langle V_{\mathcal{H}}, E_{\mathcal{H}} \rangle$ where $V_{\mathcal{H}}$ and $E_{\mathcal{H}}$ represent the set of vertices and edges respectively. Given a set of example policies $\mathbb{G} = \{G_1, G_2, ..., G_n\}$ generated by an MDP solver, the corresponding abstracted set of policies $\mathbb{G}' = \{G'_1, G'_2, ..., G'_n\}$ are found using Υ. The generalized policy graph is constructed as follows:

$V_{\mathcal{H}} = \{\bar{\mathcal{L}}_{G'_j}(v_i) \mid v_i \in V_{G'_j}, G'_j \in \mathbb{G}'\}$

$E_{\mathcal{H}} = \{\langle \bar{\mathcal{L}}_{G'_j}(v_i), \bar{\mathcal{L}}_{G'_j}(v_j) \rangle \mid \exists G'_j \in \mathbb{G}' s.t.\, v_i, v_j \in V_{G'_j}, \langle v_i, v_j \rangle \in E_{G'_j}\}$

Thus, the generalized policy $\mathcal{H}$ contains the set of all unique abstract state-action pairs seen in abstracted example policies as its vertices and set of all abstract state-action pair transitions seen in abstracted example policies as its edges.

If unrolled sequences, which are repeated sequences of state-action pairs seen in root-to-leaf paths of an abstract policy G' are seen, then a cycle is formed in place of the unrolled sequence in generalized policy $\mathcal{H}$ to capture such redundancies.

To build this graph, Algorithm 3 (Merge) is called iteratively. The Merge algorithm accepts an input example abstracted graph $G'_i \in \mathbb{G}'$ and the current generalized policy graph $\mathcal{H}$. Merge follows a greedy approach to form loops in the generalized policy.

The Merge algorithm is initially input with $\mathcal{H}$ containing no nodes and edges, i.e. $V_{\mathcal{H}} = \emptyset$, and $E_{\mathcal{H}} = \emptyset$. It computes the set of unique state-action pairs and transitions in G' as shown in Line 1-2 of Algorithm 3. Once this is done, the change-sets for nodes and edges required to be added to $\mathcal{H}$ are calculated. These are defined as the set differences shown in Lines 3-4 of Algorithm 3. If both the change-sets are found to be empty, then this indicates that the trace G' is not adding any new information to the generalized policy $\mathcal{H}$, and Merge returns 'false' to indicate this. Otherwise, the nodes and edges of $\mathcal{H}$ are updated to include all new state-action pairs and transitions captured in the node and edge change-sets. This is done in Line 7-8 of Algorithm 3. Merge returns 'true', indicating that G' added new information that was used to update $\mathcal{H}$. Note that at any point, the set of nodes and set of edges present in $\mathcal{H}$ are unique i.e. $\forall(v \in V_{\mathcal{H}})\ (v = \langle \bar{s}, \bar{a} \rangle)$; $\bar{s} \in \bar{S}$; $\bar{a} \in \bar{A}$; $\forall(e \in E_{\mathcal{H}})\ (e = \langle \langle \bar{s}_i, \bar{a}_i \rangle \langle \bar{s}_j, \bar{a}_j \rangle \rangle)$; $\bar{s}_i, \bar{s}_j \in \bar{S}$; $\bar{a}_i, \bar{a}_j \in \bar{A}$.

```
Input   : $\mathcal{H}, G'$
Output: success, $\mathcal{H}$
1 $unique\_sa\_pair = \{\mathcal{L}_{G'}(v) \mid v \in V_{G'}\}$
2 $unique\_sa\_transition = \{\langle \mathcal{L}_{G'}(v_i), \mathcal{L}_{G'}(v_j) \rangle \mid v_i, v_j \in V_{G'}, \langle v_i, v_j \rangle \in E_{G'}\}$
3 $node\_change\_set = unique\_sa\_pair \setminus V_{\mathcal{H}}$
4 $edge\_change\_set = unique\_sa\_transition \setminus E_{\mathcal{H}}$
5 if $(node\_change\_set = \emptyset) and (edge\_change\_set = \emptyset)$ then
6     return $False, \mathcal{H}$
7 $V_{\mathcal{H}} = V_{\mathcal{H}} \cup node\_change\_set$
8 $E_{\mathcal{H}} = E_{\mathcal{H}} \cup edge\_change\_set$
9 return $True, \mathcal{H}$
```

Algorithm 3: Merge($\mathcal{H}, G'$)

An example generalized policy, initially containing no nodes and edges, and trained with an input trace shown in Fig. 4.5 is shown in Fig. 4.6. I shall now present theoretical results related to the uniqueness and succinctness of the policies computed by Merge.

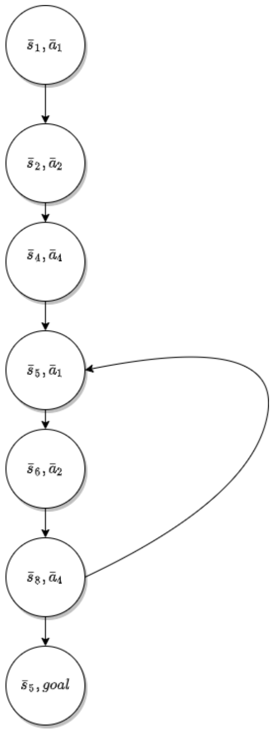


Figure 4.6: Generalized Partial Policy Graph

Theorem 1. *(Uniqueness): Merge will result in the same unique generalized policy $\mathcal{H}$ regardless of the order in which example policies are used to build it.*

Proof. Consider example traces $\mathbb{G}' = \langle G'_1, G'_2, \dots, G'_n \rangle$. Let $\mathcal{H}$ be the generalized policy formed after using Merge by using elements from $\mathbb{G}'$ in its current order, and $\mathcal{H}'$ be the generalized policy formed using elements from $\mathbb{G}'$ in a different order.

The set of all abstract state-action pairs observed in example traces $\mathbb{G}'$ is given by the set $I_{pair} = \{\bar{\mathcal{L}}_{G'_j}(v_i) \mid v_i \in V_{G'_j}, G'_j \in \mathbb{G}'\}$. The merge algorithm assigns the vertices of the generalized policy graph it forms to this set. This means that $V_{\mathcal{H}} = I_{pair}$ and $V_{\mathcal{H}'} = I_{pair}$. Hence, $V_{\mathcal{H}} = V_{\mathcal{H}'}$.

Similarly, the set of all abstract state-action transitions in example traces is given by the set $I_{transition} = \{\langle \bar{\mathcal{L}}_{G'_j}(v_i), \bar{\mathcal{L}}_{G'_j}(v_j) \rangle \mid \exists G'_j \in \mathbb{G}' s.t. \, v_i, v_j \in V_{G'_j}, \langle v_i, v_j \rangle \in E_{G'_j}\}$. The merge algorithm assigns the edges of the generalized policy graph it forms to this set. So, $E_{\mathcal{H}} = I_{transition}$ and $E_{\mathcal{H}'} = I_{transition}$. Hence, $E_{\mathcal{H}} = E_{\mathcal{H}'}$.

Since $V_{\mathcal{H}} = V_{\mathcal{H}'}$ and $E_{\mathcal{H}} = E_{\mathcal{H}'}$, $\mathcal{H} = \mathcal{H}'$ is true. Thus, the merge will result in the same unique generalized policy $\mathcal{H}$ regardless of the order in which it example policies are used to build it. $\square$

Theorem 1. *(Succinctness): The generalized policy $\mathcal{H}$ constructed using canonical abstraction by Merge from a set of example traces $\mathbb{G}'$ has bounded $|V_{\mathcal{H}}|$ and $|E_{\mathcal{H}}|$, and captures all unique state-action pairs and transitions seen in $\mathbb{G}'$.*

Proof. Consider example traces $\mathbb{G}' = \langle G'_1, G'_2, \dots, G'_n \rangle$. The set of unique abstract state-action pairs and transitions seen in example traces is given by I_{pair} and $I_{transition}$ respectively as shown in Theorem 1. The conditions $|I_{pair}| \leq |\bar{S} \times \bar{A}|$ and $|I_{transition}| \leq |\bar{S} \times \bar{A} \times \bar{S} \times \bar{A}|$ hold. Since I_{pair} and $I_{transition}$ contain only unique elements, $|I_{pair}|$ and $|I_{transition}|$ represent the minimum possible number of nodes and edges required in a generalized policy graph to capture all abstract state-action pairs and transitions respectively.

All nodes in the generalized policy formed by merge represent unique abstract state-action pairs seen in $\mathbb{G}'$. So, $|V_{\mathcal{H}}| = |I_{pair}|$. Similarly, all edges in the generalized policy represent unique abstract state-action transitions seen in $\mathbb{G}'$.
So, $|E_{\mathcal{H}}| = |I_{transition}|$. Hence, even if an infinite number of examples are used to learn the generalized policy $\mathcal{H}$, the number of nodes and edges of $\mathcal{H}$ are bounded by the number of unique abstract state-action pairs and transitions seen in $\mathbb{G}'$. $\square$

The time complexity of the Merge algorithm to merge a generalized policy $\mathcal{H}$ with a trace G' is $O(max(|V_{\mathcal{H}}|, |V_{G'}|) + max(|E_{\mathcal{H}}|, |E_{G'}|))$.

4.2 Generalized Policy Instantiation

After learning a generalized policy $\mathcal{H}$, it is used to find solutions to some problem instance Γ. This requires information that can be used to find concrete policies for problems of larger sizes if the abstracted solution to the problem is encompassed by the generalized policy.

A generalized policy instantiation problem is the tuple $\Omega = \langle \Gamma, \mathcal{H} \rangle$ where Γ is an MDP and $\mathcal{H}$ is a generalized partial policy.

The solution to a Ω, is a finite, connected, directed graph $C = \langle V_C, E_C \rangle$, where V_C and E_C represent the set of vertices and edges of C. C represents a partial policy graph that is a solution for the MDP Γ.

A domain file and a problem file generated using the problem generator are used to generate Γ. The processes involved in this section are described in Fig. 4.7.

Application of an action $a \in A$ from state $s \in S$ requires that $s \models pre(a)$. Each action can have multiple possible effects, each of which are associated with an effect list ϵ_a. The action a_e represents a sub-action of action a, for which $pre(a) = pre(a_e)$ and $\epsilon_{a_e} = \langle \langle \mathfrak{p}_e, e \rangle \rangle$, $\langle \mathfrak{p}_e, e \rangle \in \epsilon_a$, where $\mathfrak{p}_e$ represents the probability with which an action a when executed changes the current state using the effect set e, and is the

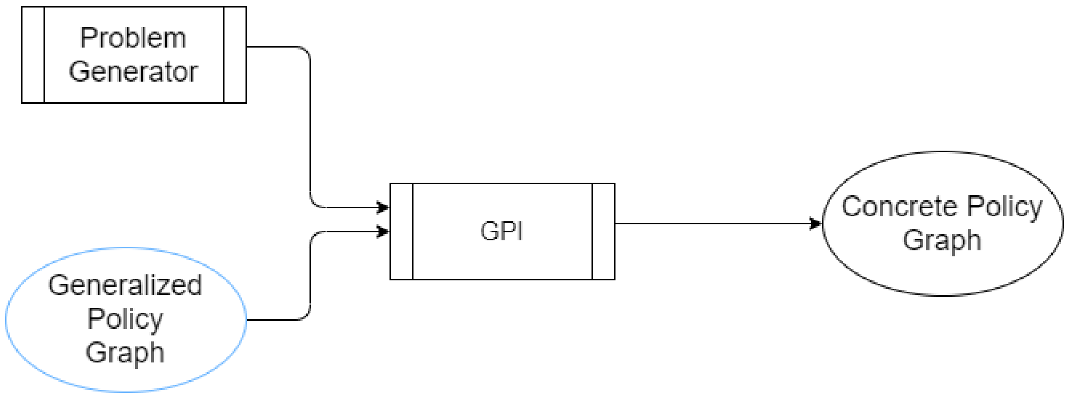


Figure 4.7: Learning Generalized Policies

probability associated with sub-action a_e. Each effect e can be split into its component add-effects which are literals added to the current state, and delete-effects which are literals deleted. An action a_e when executed at a state $s \in S$ results in a new state $s' = (s \setminus e^-) \cup e^+$, $s' \in S$. This operation is denoted by $s' = a_e(s)$.

Algorithm 4 (GPI) uses the generalized policy graph $\mathcal{H}$ to direct the policy search to find a solution for a given problem instance Γ. Each stochastic action is treated as a set of deterministic sub-actions, as described above. Portions of source code developed by (Steinmetz *et al.* (2016)) were used to implement this. Upon the execution of a stochastic action on a state $s \in S$, GPI calculates a list of successor states by executing all sub-actions, and each successor state can be reached from the current state based on the probability corresponding to that sub-action.

Each node $n \in V_C$ corresponds to a concrete state-action pair $\langle s_n, a_n \rangle$, $s_n \in S, a_n \in A$. Initially, a single node $n_c \in V_C$, $s_{n_c} = s_{init}$ is initialised. n_c represents the current node being explored in C. GPI attempts to find a sequence of such deterministic actions that can be applied, starting from the initial state s_{init} to reach a state satisfying the goal condition $\mathcal{G}$. During each step, a corresponding current abstract node $h_c \in V_{\mathcal{H}}$ is maintained, where $s_{h_c} = \bar{s}_{n_c}, s_{h_c} \in \bar{S}, s_{n_c} \in S$. The current abstract action $a_{h_c} \in \bar{A}$ is used in filtering out the set of actions to be explored from state

s_{n_c}, of which the one executed is $a_{n_c} \in A$. In this way, the generalized policy $\mathcal{H}$ helps direct the search. Upon each execution, a new node n_n, corresponding to the next node to be explored is created in C, with $s_{n_n} = a_{n_c}(s_{n_c})$. An edge is created between n_c and n_n in C. Once a sequence of actions from the initial to the goal state is found, the sequence is retraced from goal to root node in C containing the initial state. A check is done in each step of this retrace to see if all possible effects for the corresponding stochastic action have been explored with their corresponding sub-actions. If not, it explores one of the unexplored effects.

Once a goal state is reached through this contingent branch, retracing is done again, and the algorithm continues to execute until all the action effects have been explored for all nodes in C. Every time a goal is reached, all ancestors of the goal node are marked to indicate that they have a path that leads to a goal node. The inputs to this algorithm are the generalized policy $\mathcal{H}$, a concrete policy graph C with the current concrete node n_c for which $s_{n_c} = s_{init}$, an abstract node h_c called the current abstract node which satisfies $s_{h_c} = \bar{s}_{init}$ and $\mathcal{G}$ which is the goal condition for the problem.

```
Input  : $C, \mathcal{H}, n_c, h_c, \mathcal{G}$
Output: $success, C$
1 if $(s_{n_c} \models \mathcal{G}) \lor ((h_c\ is\ a\ dead\text{-}end\ node) \land (C.has\_marked\_ancestor(n_c)))$ then
2     C.mark_ancestors($n_c$)
3     return True
4 next_act_list = $\{a_i | (a_i \in A) \land (\bar{a}_i = a_{h_c}) \land (s_{n_c} \models pre(a))\}$
5 if $|next\_act\_list| = 0$ then
6     return False
7 for $a$ in heuristic(next_act_list) do
8     C.create_node($n_n$)
9     if $\neg n_c[\varepsilon_a]$ then
10        $n_c[\varepsilon_a] = \varepsilon_a$
11    $e = n_c[\varepsilon_a].pop()$
12    $a_e$ =get_sub_action($a, e$)
13    $s_{n_n} = a_e.apply(s_{n_c})$
14    if $\exists n_v \in V_C\ s.t.\ (s_{n_v} = s_{n_n}) \land C.is\_marked(n_v)$ then
15        C.add_edge($n_c, n_v$)
16        C.mark_ancestors($n_c$)
17        if $|n_c[\varepsilon_a]| = 0$ then
18            return True
19        else
20            goto line 30
21    C.add_edge($n_c, n_n$)
22    for $h_n$ in abstract_heuristics($C.get\_valid\_children(\mathcal{H}, s_{n_n})$) do
23        ret = GPI($C, \mathcal{H}, n_n, h_n, \mathcal{G}$)
24        if $ret$ then
25            break
26    if $ret \land |n_c[\varepsilon_a]| = 0$ then
27        return True
28    if $\neg ret$ then
29        C.remove_edge($n_c, n_v$) and continue to next iteration
30    if GPI($C, \mathcal{H}, n_c, h_c, \mathcal{G}$) then
31        return True
32    C.check_and_unmark_ancestors($n_c$)
33    C.remove_outgoing_edges($n_c$)
34 return False
```

Algorithm 4: GPI

Line 1-3 describe the cases where a goal condition is found. If the current state $s_{n_c} \models \mathcal{G}$, then the search algorithm returns True to the call stack. The second portion of the condition in Line 1 handles the scenario where a path to a goal from the initial state is already found, and an alternate action effect is being explored, during which a dead-end node was encountered. Since a goal state was reached, there must be some ancestor of the current node that is marked as a goal node. So C.has_marked_ancestor(n_c) is True. In such a situation, the call to GPI returns True. Policies where such situations are encountered represent improper policies. They can occur when the $\Gamma \in$ fSSPUDE MDPs, where all solutions contain dead ends. They can also occur in cases where $\Gamma \in$ SPADE MDPs when poor training examples containing improper policies are used to form the generalized partial policy. Since LAO* produces optimal proper policies for both SSPs and SSPADEs, the only scenario where dead nodes are encountered is when the MDP problem is a subset of fSSPUDEs. A marked node indicates that it contains a valid path to a goal state. Before returning to the call, the current node and its ancestors are marked, indicating that these nodes have a valid path to the goal. The function call C.mark_ancestors marks all the ancestor nodes of n_c to indicate that they have a valid path to the goal. Note that nodes marked by GPI as dead-ends are those that contain states from which GPI could not find paths to the goal using $\mathcal{H}$. So, these nodes are called *partial dead-ends.*

During the exploration of a state, a list of applicable actions, called 'next_act_list', on the given state is calculated. These actions are filtered by using the current abstract action a_{h_c}, as shown in Line 4 of Algorithm 4. If no applicable action is found, return False. This indicates that further exploration directed by $\mathcal{H}$ does not lead to a goal state.

When actions in next_act_list were explored in any random order in experiments conducted, GPI did not yield results within reasonable time frames. Directing GPI with the help of heuristics that can order the set of actions to be explored significantly improved the performance. In this heuristic, each action in next_act_list is scored based on the number of matching predicates from the goal condition and the number of predicates in initial states required to be removed to reach the goal state. Actions that the heuristic predicts have higher chances to lead to the goal state are given higher scores and appear earlier in next_act_list. In case all actions have the same preference, which can arise in situations where that action does not directly contribute to predicates seen in the goal state, a look-ahead is performed where the scoring of future possible actions using the heuristic described above is done. This is used to rank actions in *next_act_list*. Based on trial and error, looking ahead with a depth of 5 actions worked suitably for most domains. The resulting actions are explored in the order assigned by the heuristic. The heuristic used makes GPI analogous to many non-interleaved planners, which are susceptible to Sussman anomaly.

When exploring each action a, a new node n_c is created and added to V_C. This is done in Line 8. If action a is being explored for the first time, a list maintaining the effect list of a is added to n_c. If such a list already exists, it means that a path was found from the current node to a terminal node, and the current node is being revisited during retracing. An effect e is popped from the effect list, and GPI explores if valid paths to the goal exist with this contingency. In lines 12-13, the sub action corresponding to a with effect e is found using 'get_sub_action'. Then, the next state s_{n_n} is calculated. If some node $n_v \in V_C$ with a path to a terminal node, and containing a state $s_{n_v} = s_{n_n}$ exists, then an edge is added from the n_c to n_v. The ancestors of n_c are then marked. If any sub-action corresponding to a remains unexplored, a recursive call to GPI is made to explore all such sub-actions. This will be explained in detail later.

If the condition in line 14 is not true, an edge is added from n_c to n_n, and $\mathcal{H}$ is consulted again to find the next list of abstract nodes corresponding to n_n. The list of child nodes of h_c contains possible candidates for the next abstract node h_n. This node must satisfy $s_{h_n} = \bar{s}_{n_n}$. Another heuristic is computed to score all possible next abstract nodes. This is done using the heuristic function described before over all possible next abstract actions. Reordering the abstract nodes whose corresponding abstract action has the highest score in an earlier position in this list also results in a significant reduction in search time required by GPI to find solutions. This is represented by *abstract_heuristics*. GPI is then called with parameters C, H, n_n, h_n, and g. If the variable *ret* is assigned true, that means a goal is found from node n_n while using h_n as the next abstract node, and we do not explore paths using a different next abstract node. If not, *ret* is assigned false. If *ret* is assigned true and no further stochastic effects of a were remaining to be explored, return true to the GPI call, indicating a path from n_c exists to reach the goal. Note that if true is returned, n_c has already been marked. If *ret* is assigned false, the edge from c_n to c_c is deleted, and the control skips to the next iteration of the for loop, where a new node is created, and other actions with lower ranks in *next_act_list* are explored.

If true is returned, and all possible stochastic effects were not explored, then GPI is recursively called with the same parameters as the current call to explore these effects. This call will explore the next effect in the effect list ϵ_a. The condition on Line 9 will no longer hold, and the next effect to be explored is given by Line 11. Subsequent sub-calls to GPI made by this call will explore all effects in ϵ_a, and delete it from ϵ_a with each exploration. If this call returns true, that means n_c has a path to reach a terminal state regardless of which action effect of a is observed. If this call returns false, this indicates not all contingencies of a could not be resolved from the current node. This also indicates there were no stop nodes in all the descendants of h_c, and hence no improper policy was permitted either. In this case, ancestors of n_c, which do not have alternate paths to the goal, are un-marked. All outgoing edges from n_c are severed, and control flows to the next iteration of the for-loop.

For b applicable actions each with c effects found during each call of GPI, for a solution found at depth d, the time complexity of GPI is $O((bc)^d)$.

Chapter 5

EXPERIMENTS

This section is divided into two parts. In the first portion, the different benchmark discrete MDP domains on which empirical evaluations of GPI's performance against state-of-the-art discrete MDP solvers were performed are described. GPI was incorporated in the existing anytime stochastic task and motion planning framework developed by (Shah *et al.* (2020)).

5.1 Domains

The methodology described in this work was compared against state-of-the-art discrete MDP solvers, namely LAO* and LRTDP. (Bonet and Geffner (2003)). Both LAO* and LRTDP employ heuristic search DP algorithms to find an optimal partial policy rooted at the initial state for an MDP problem. The different discrete MDP domains used for evaluation are described in each subsection. Although GPI is not guaranteed to produce optimal partial policies, it does find a proper policy in case of problems belonging to the SSP and SSPADE subclasses (given that training problems used to form the generalized policy for SSPADE were optimal) if the generalized policy encompasses their goal condition. In the case of fSSPUDE problems, off-the-shelf MDP solvers and GPI produce improper policies as solutions. Note that solutions to an MDP can be found by GPI only if the goal condition $\mathcal{G}$ is encompassed by the generalized partial policy $\mathcal{H}$.

5.1.1 Keva

Keva planks are laser cut planks made of wood that are sold as toys to kids. Every plank is identical to another, with minimal variance in size. Using these planks, multiple 3D structures can be built. Solutions to problems in this are high-level plans a robot has to execute to build such structures. The problems considered focus on building towers and pi structures. To build a structure, a human places a plank in one of two locations with a probability of 0.6 of placing it in location1 and 0.4 of placing it in location2. The robot then picks up the plank from the location it was placed in and places it in the appropriate configuration as described by the goal condition to build the structure. Each plank can be placed either on the table, on another plank, or two planks. All placements are reversible, i.e., the robot can pick a plank after it was placed and place it elsewhere. The MDP problems corresponding to this domain fall under the SSP sub-class of MDP problems.

5.1.2 Delicate-Can-World

Multiple cans are present on a cluttered table, and the goal is to pick a specific can. The cans are divided into two types: delicate and non-delicate. A robot with a single gripper is used to manipulate the environment. Since the table is cluttered, the robot's goal cannot be reached as other cans may obstruct the robot from picking the goal can. The problems in this domain are such that all the cans are obstructing the goal can and hence have to be moved. The non-determinism in this domain originates in the pick action. If a delicate can is picked, it has an 80% probability of being crushed. A non-delicate can has a 10% probability of being crushed when picked. If a can is crushed, it has to be thrown in the dustbin. Otherwise, it is placed in another region of the table where it is guaranteed not to obstruct the goal can. The delicate can requires three units of effort to clean up after being crushed, while the non-delicate one requires one unit. The MDP problems corresponding to this domain fall under the SSP sub-class of MDP problems.

5.1.3 Can-World

This domain models a similar environment as the DelicateCanWorld domain. Here, not all cans present on the table necessarily obstruct the robot from picking the goal can. The stochasticity in this domain originates from the pick action, with a can being crushed on the execution of pick with a 10% probability. If a can is crushed, it is immediately thrown into a dustbin. The MDP problems corresponding to this domain fall under the SSP sub-class of MDP problems.

5.1.4 Gripper

There are two rooms ‘rooma’ and ‘roomb’. Multiple balls are placed in each room. Each ball has an intended target room. A robot with gripper(s) is used to pick a ball(s), move to a room, and place it. The stochasticity in this domain lies in the pick action, where there is a 20% probability that the ball is dropped when a pick action is executed. For the Gripper domain, the problems generated contain either a robot with a single gripper or one with two grippers. The MDP problems corresponding to this domain fall under the SSP sub-class of MDP problems.

5.1.5 Hanoi

Hanoi is a well-studied problem in AI. There are three pegs and multiple discs on the first peg, each with decreasing disc sizes with the largest disc placed at the bottom. The goal is to move all the discs from peg1 to peg3 without placing a larger disc on a smaller disc at any step. The only action in this domain is move, where a disc is moved from one position to another. The inherent stochasticity introduced in this domain is that the execution of move action can accidentally break the whole setup with a probability of 1/100000. The MDP problems corresponding to this domain fall under the fSSPUDE sub-class of MDP problems, containing only improper policies as solutions.

5.1.6 File-World

In this domain, there are two kinds of objects: folders and files. The goal condition requires each file to be filed inside one of the folders. To file a file, firstly, the file type, which indicates which folder it should be filed in, is found by executing the get_file_type action. This action is stochastic, where each non-deterministic effect sets the file type as one of the available folders with a probability of $1/\#(folders)$. For the problems considered in this work, $\#(folders) = 2$. Once the type of the file is found, get_folder is executed. Then the file is filed in the folder, and the folder is returned. The goal condition specifies that all files should be filed. This domain is a corner case of the same name domain, considered in IPC2004 probabilistic planning competitions. The MDP problems corresponding to this domain fall under the SSPADE sub-class of MDP problems.

5.1.7 Rover

The planetary rovers domain inspires this domain. A map defined by waypoints is given where each waypoint can have multiple incoming and outgoing paths from and to other waypoints, respectively. Objectives are images that need to be captured, and each objective can be completed only from a specific waypoint. There are multiple samples distributed across waypoints, each of which has to be picked and dropped into a drop location individually. The rover can move between two waypoints if a path exists between them. The pick_sample is a stochastic action that successfully picks the sample with an 80% probability and fails with a 20% probability. The problems considered contain fully connected maps which have a total of 3 or 4 waypoints. Each problem contains only two objectives. The number of samples that need to be picked is varied across problems. The MDP problems corresponding to this domain fall under the SSP sub-class of MDP problems.

5.1.8 Schedule

This domain is a simplified version of the schedule domain considered in IPC2006 probabilistic planning competitions.

Problems involve multiple packets that need to be served by a server. Initially, the server is in the Arrivals_and_updating phase. Each time a packet arrives at a server, it is marked as processed, and with a probability of 47/50, it is assigned a packet class and put on the queue. With a probability of 3/50, it is not assigned a class. Then, the server switches to the Cleanup_and_service phase. If a packet was assigned a packet class, it is served. Otherwise, it is dropped and marked unprocessed. This allows the server to receive the packet again and process it for arrival. The goal is to serve all incoming packets. The problems considered have a different number of packets in them. The MDP problems corresponding to this domain fall under the SSP sub-class of MDP problems.

5.2 Anytime Integrated Task And Motion Planning

The methodology developed as a part of this thesis is integrated with the anytime integrated task and motion motion planning framework developed by (Shah *et al.* (2020)). The ATM-MDP algorithm developed by (Shah *et al.* (2020)) to accomplish the task is used to build a free-standing Keva tower structure with 12 levels. The authors of Shah *et al.* (2020) use the LAO* algorithm to compute a high-level policy to perform this task. This is replaced with the GPI method developed in this work for this experiment. The OpenRAVE simulator (Diankov and Kuffner (2008)) is used for performing this planning experiment along with its collision checkers. The CBiRRT (Berenson *et al.* (2009)) implementation from (Koval (2014)) was used for motion planning. A model of the Yumi IRB14000 dual hand robot is used as the autonomous agent acting in the stochastic environment for this experiment. To find inverse kinematic solutions, Trac-IK (Beeson and Ames (2015)) is used. The goal structure required to be built is described in a Collada file, as shown in Fig. 5.1. Pose generators were written to use this reference structure during refinement performed by the ATM-MDP algorithm to build the desired structure. The environment contains two tables, one on which the Yumi robot is placed, and another where the desired structure is built. On this second table, two stations representing the two different locations where a human can place a plank are present. The robot must pick the plank placed in any of these regions by the human and place it appropriately to build the structure.

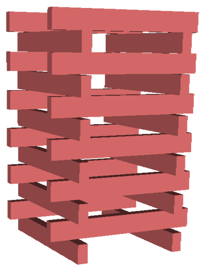


Figure 5.1: Keva 12-Level Tower Structure

Chapter 6

RESULTS

6.1 Domains

Empirical evaluations were made for determining which problems should be used for training the generalized policy. After the generalized policy was formed, the run times of GPI were compared against that of baselines, namely LAO* and LRTDP, to solve examples from a test set of discrete MDP problems. Table 6.1 describes the number of training and testing problems used for each domain.

6.1.1 Training

Experiments were performed for each benchmark domain to determine the set of training examples to be used. Using the problem generator for each domain, example problems were generated. The LAO* algorithm was used to find the optimal partial policy graph for each problem. Problems with an increasing number of objects in them were solved in increasing order, and their traces were found. The Merge algorithm was used to incrementally build the generalized partial policy using these traces in the order they were found. It was observed that the algorithm reported successfully using these examples to build generalized policies only for problems with object counts up to some finite value. This value is called maximum training object count (MTOC). After this point, all other problems with larger object counts did not contribute to the learning of the generalized policy. All problems with object count lesser than or equal to MTOC were used for training. MTOC is represented by the black vertical dotted line in Fig 6.1 ([I-IX]-a) for each domain.

Domain	Training	Testing
Can-World	9	135
Hanoi	4	32
Rover	10	30
Delciate-Can-World	45	105
Keva	9	23
Schedule	5	40
File-World	4	36
Gripper	10	48

Table 6.1: Total Problems Used For Training And Testing

6.1.2 Testing

Once the generalized policy for a domain is generated, a set of problems for that domain are created, with each problem's object count falling within a range of values. This is a domain-specific configuration and is illustrated by the range of values shown in the x-axis 'Objects (#)' in Fig 6.1 ([I-IX]-a) for each domain. Run times for GPI and baseline approaches were compared. The cutoff time for the baselines LAO* and LRTDP were set to 600s. The cutoff time for GPI was set to 200s, as it finds greedy solutions to problems, unlike the baselines, which attempt to find optimal solutions.

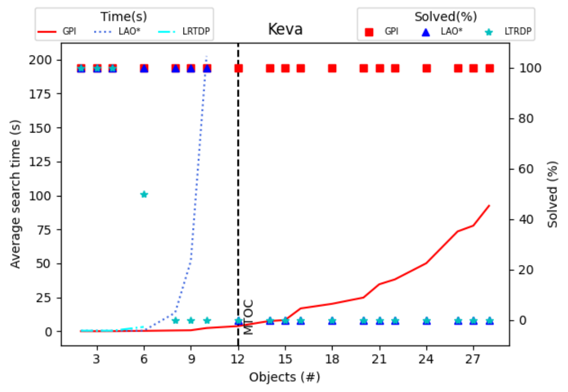
Fig 6.1([I-IX]-a) compares the average run-time for a successful search between GPI and baseline approaches and plots it against the number of objects present in the problem. For example, in Fig 6.1 (III-a) consider a point on the x-axis where $Object(\#) = 9$. The corresponding y-value of a curve represents the average time taken to successfully solve all Delicate-Can-World problems, which had object counts equal to 9. Object counts in this domain correspond to the total number of cans. Different problems with object counts equal to 9 contain a different number of delicate cans ranging from 1 to 8 present in the problem. Cases where an approach failed to find the solution for a problem are not included in the curves corresponding to that approach.

Fig 6.1 ([I-IX]-a) also compares the success rates in the percentage of total problems solved for all three approaches, each of which is represented by three different scatter plots in the figure. It can be observed that for problems in our test suite, GPI solves 100% of problems for all object counts, for all the benchmark domains considered, while the LAO* and LRTDP algorithms fail to do so in many cases. These plots showcase that the generalized policies formed using the methodology developed in this work generalize well to unseen problems of larger sizes. It can also be observed that except for the File-World and Hanoi domains, the average search times for problems solved by GPI is much lower than that of LAO* and LRTDP. Looking at Fig 6.1 (III-a) reveals that the average time taken to solve a problem appears to be an exponential function of the number of objects present in the problem. So, to solve a problem instance with an object count of 14 in this domain, it appears that the time taken to compute a solution by LAO* and LRTDP would be well above the cutoff time of 600s. However, GPI can find a solution in less than 60s on average. This shows that GPI is a very attractive candidate in situations where optimality can be traded for the reduced time taken to compute sub-optimal solutions.

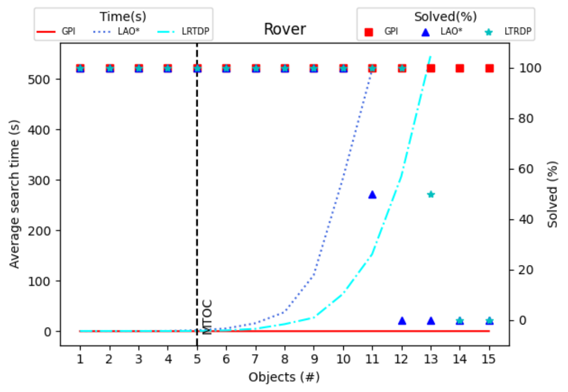
Fig 6.1 ([I-IX]-b) shows the cumulative planning time vs percentage of solved problems for GPI and the baselines. As the amount of time increases, the number of problems solved by each approach increases monotonically. It can be seen that GPI reaches 100% faster than both LAO* and LRTDP except for problems in the File-World domain.

Fig 6.1 ([I-IX]-c) shows the serialized planning time vs percentage of solved problems, where each problem is solved sequentially after the previous one is solved. In cases where a solution was not found, the time taken for that computation is not included in this plot. The total time taken to form the generalized policy, which includes LAO* search time for each example problem, time taken to find the trace, and time taken to merge it with the generalized policy, is represented by the black vertical line. The time taken by GPI to solve a problem is offset by this training time taken. The total training time also includes problems whose solutions were not used in building the generalized policy because it contained redundant information. Comparison of baselines LAO* and LRTDP with GPI, and observe that GPI can solve problems faster even after including the training time offset, except for the File-World domain.

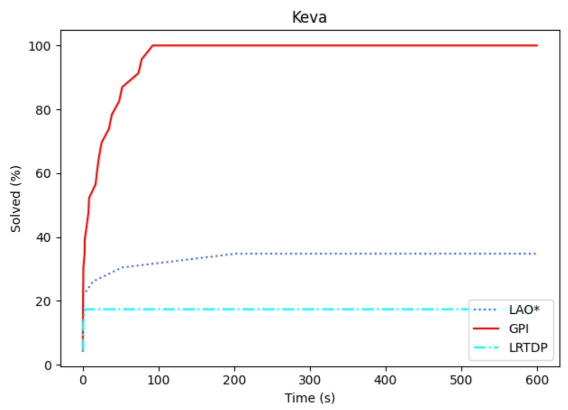
The speed of computation for GPI is highly susceptible to the performance of the heuristic. Although GPI performs well in most cases, it is susceptible to Sussman anomaly, which is the reason why it performs poorly in the case of the Hanoi and File-World domains. If the recursive call at an early stage in the search leads to some states that do not have a path to a goal, GPI will end up exploring all those states before trying another path. This is the biggest drawback of the methodology presented in this work.



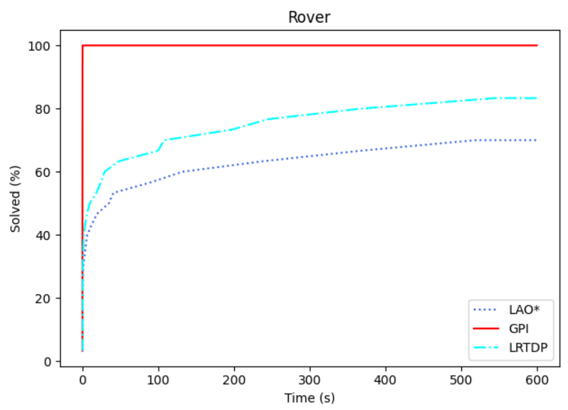
(I-a) Object vs Time(s) vs Solved(%)



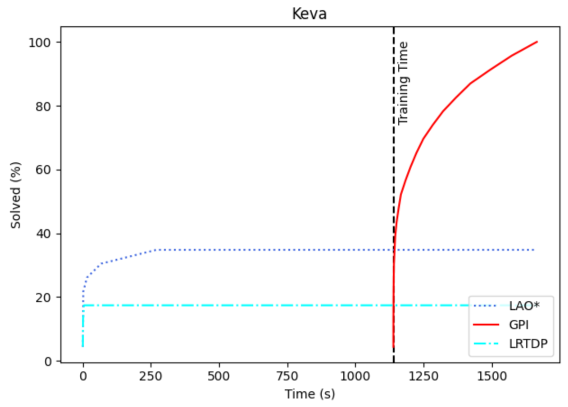
(II-a) Object vs Time(s) vs Solved(%)



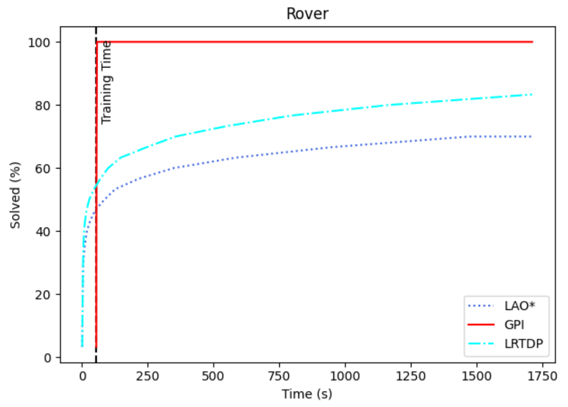
(I-b) Cumulative Time(s) vs Solved(%)



(II-b) Cumulative Time(s) vs Solved(%)

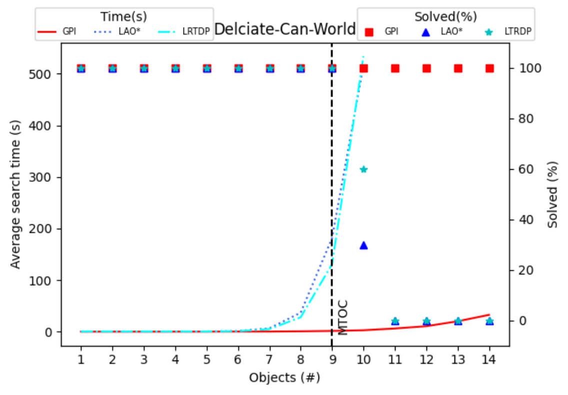


(I-c) Serialized Time(s) vs Solved(%)

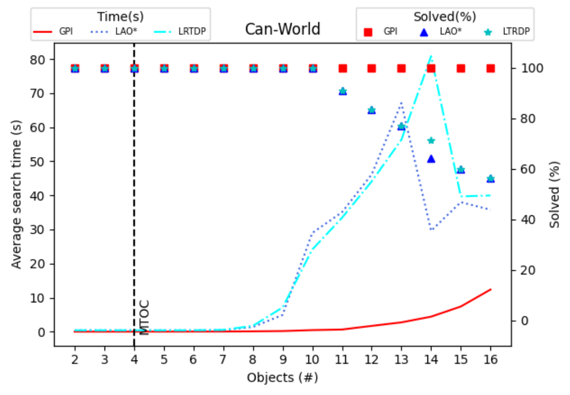


(II-c) Serialized Time(s) vs Solved(%)

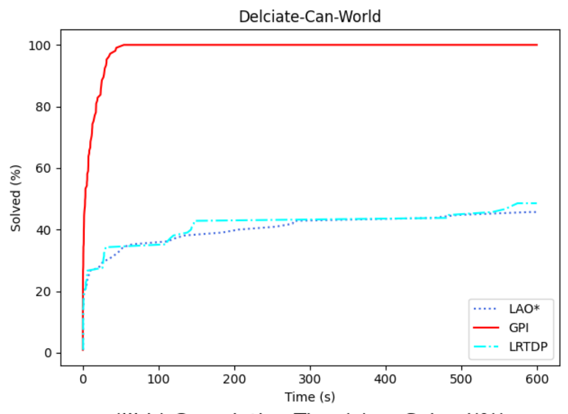
Figure 6.1: Results For (I) Keva And (II) Rover



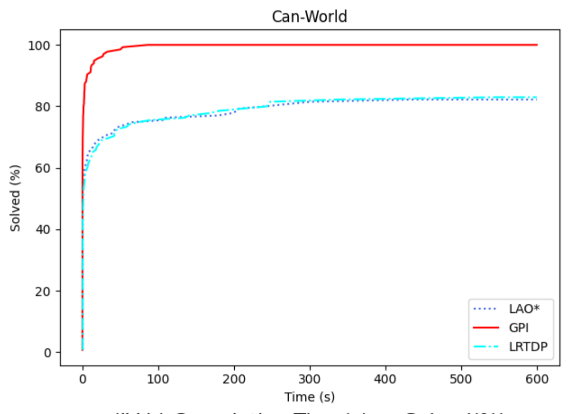
(III-a) Object vs Time(s) vs Solved(%)



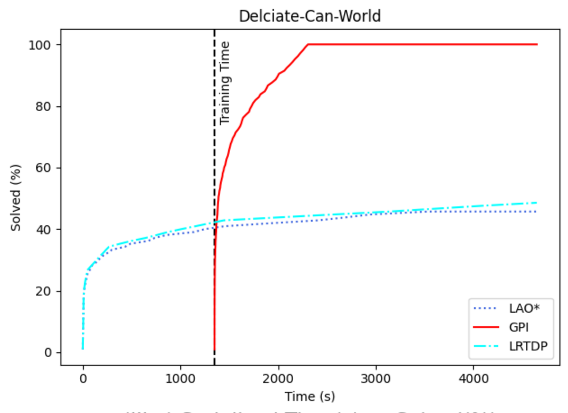
(IV-a) Object vs Time(s) vs Solved(%)



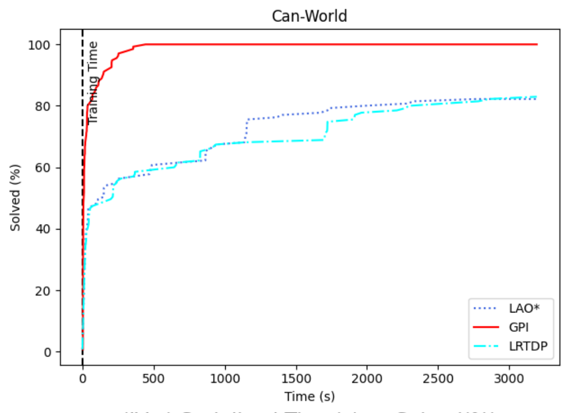
(III-b) Cumulative Time(s) vs Solved(%)



(IV-b) Cumulative Time(s) vs Solved(%)

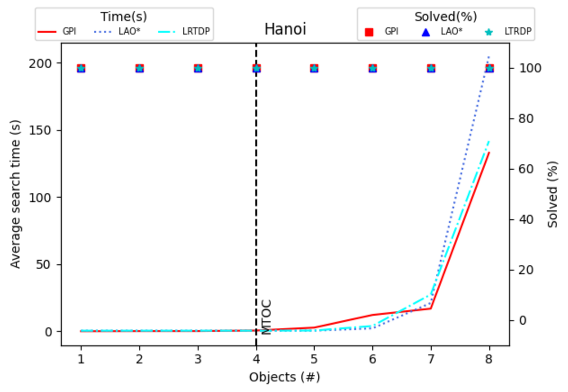


(III-c) Serialized Time(s) vs Solved(%)

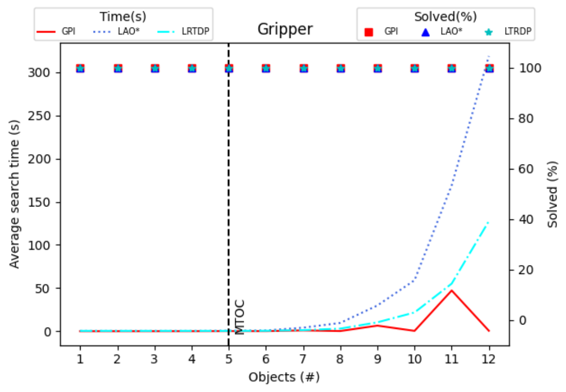


(IV-c) Serialized Time(s) vs Solved(%)

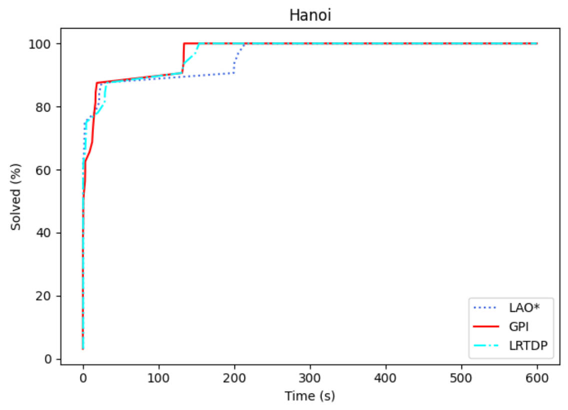
Figure 6.1: Results For (III) Delicate-Can-World And (IV) Can-World



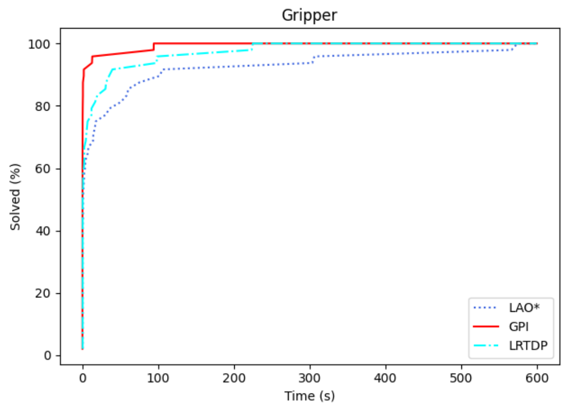
(V-a) Object vs Time(s) vs Solved(%)



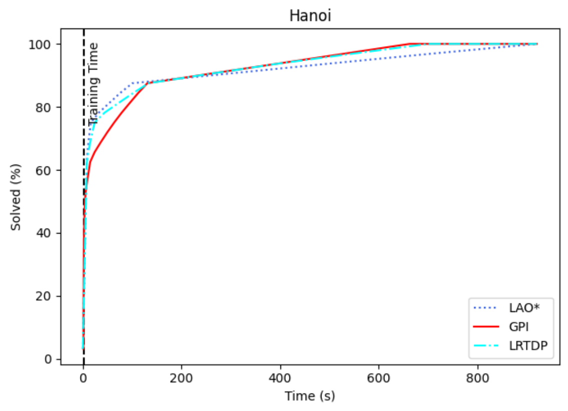
(VI-a) Object vs Time(s) vs Solved(%)



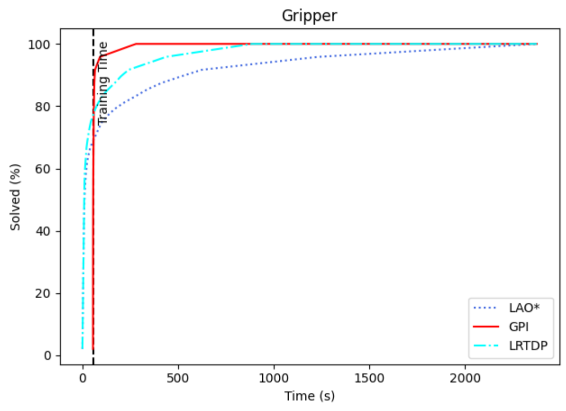
(V-b) Cumulative Time(s) vs Solved(%)



(VI-b) Cumulative Time(s) vs Solved(%)

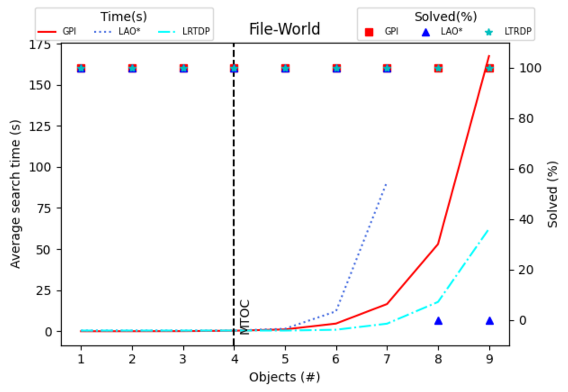


(V-c) Serialized Time(s) vs Solved(%)

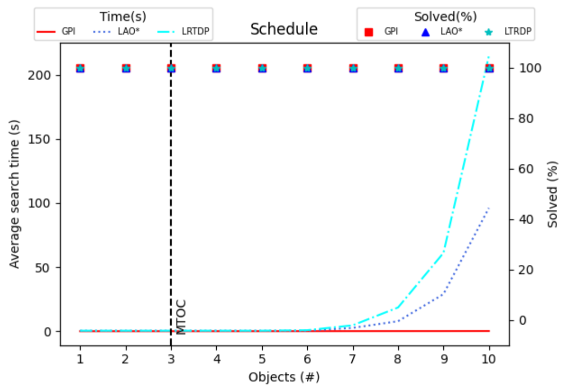


(VI-c) Serialized Time(s) vs Solved(%)

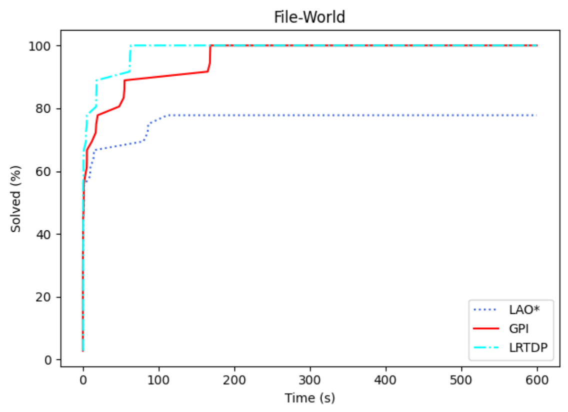
Figure 6.1: Results For (V) Hanoi And (VI) Gripper



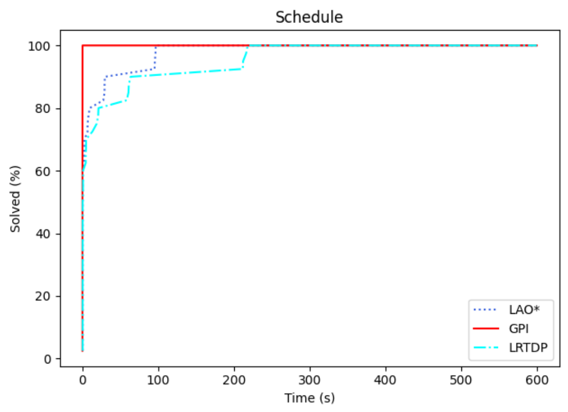
(VII-a) Object vs Time(s) vs Solved(%)



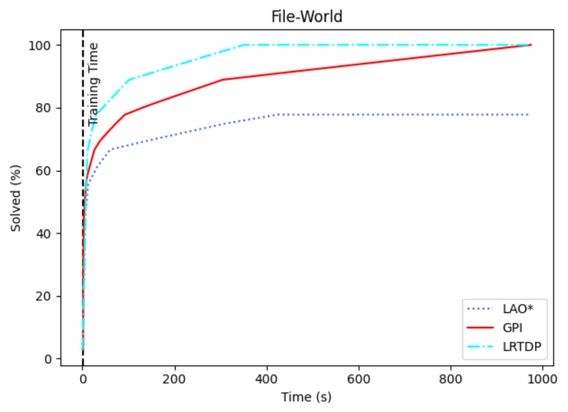
(VIII-a) Object vs Time(s) vs Solved(%)



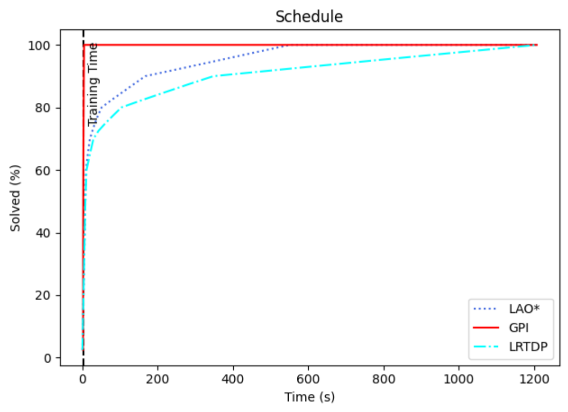
(VII-b) Cumulative Time(s) vs Solved(%)



(VIII-b) Cumulative Time(s) vs Solved(%)



(VII-c) Serialized Time(s) vs Solved(%)



(VIII-c) Serialized Time(s) vs Solved(%)

Figure 6.1: Results For (VII) File-World And (VII) Schedule

6.2 Anytime Integrated Task And Motion Planning

The stochastic task and motion planning problem defined in the experimental setup is attempted to be solved. Using GPI, the high-level policy is computed quickly. Planks are placed randomly at one of the two stations by the human in this experiment. The ATM-MDP algorithm can compute low-level motion plans required to successfully build the tower structure, despite the stochastic nature in which the human places the planks in one of the two stations present on the table. When the LAO* algorithm is used to compute the high-level policy, it fails to do so even after 1200s. Refinement is not even possible without a high-level plan in place. Waiting for an optimal solution is not desirable if a task can be accomplished while using a sub-optimal solution, that finishes quicker when compared to the time taken to compute the former. This illustrates situations where computing a quick solution that may be sub-optimal is extremely useful. A snapshot of the environment shows the successfully built structure in Fig. 6.2.

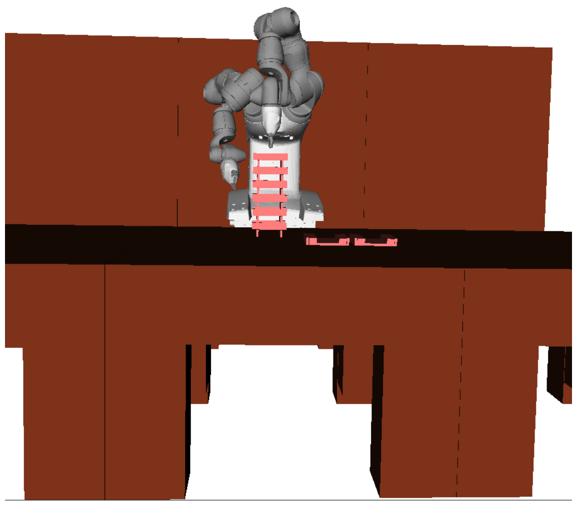
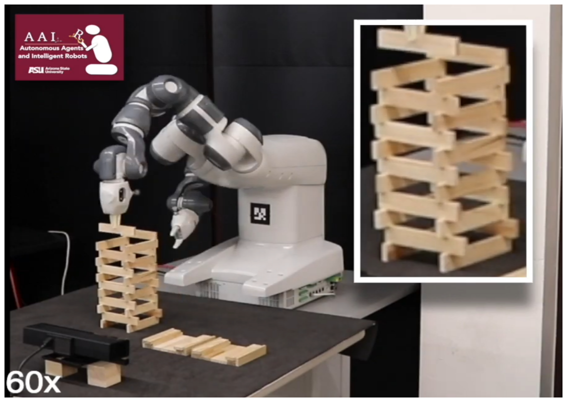
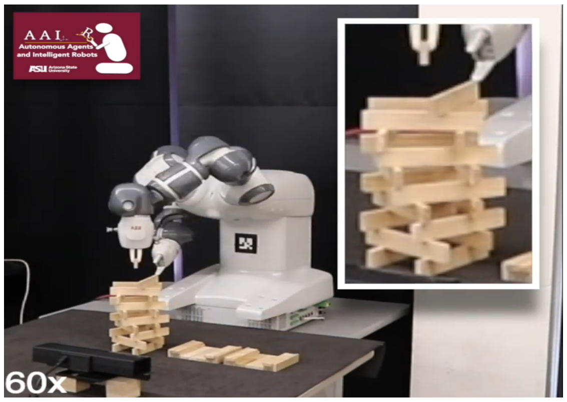


Figure 6.2: Completed 12-Level Keva Structure

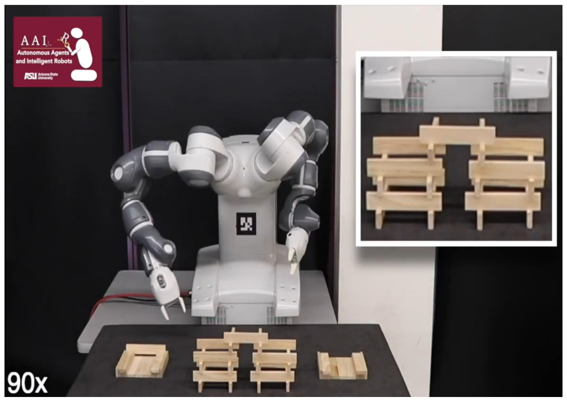
A preliminary version of this algorithm, which was used in (Shah *et al.* (2020)) was used to conduct real-world experiments of building Keva structures. An open-source controller interface (Mahler and Liang (2017)) was used to control the Yumi IRB 14000 robot. The trajectories calculated in the OpenRave simulator during the execution of the ATM-MDP algorithm are processed and sent to this interface to control the real robot. The real-world experimental setup consists of two tables, one with the Yumi, and the other with two plank stations where planks can be placed by humans. The desired structure is built on this second table. The placement of a plank by the human results in an uncertainty in the state. To eliminate this uncertainty, the environment is sensed to determine the location where the human placed the plank using a camera. A background subtraction implementation from OpenCV is used in this determination. Real-world executions are very difficult to get right as a lot of errors are introduced during the low-level state estimation done while constructing simulated environments. Collisions of the robot hand with the table are a common occurrence. To limit the chance of collisions, all actual pick and place operations are enveloped between their corresponding 'pre' and 'post' operations, where the pose of the gripper is actually a few centimeters away from the intended pick/place pose. Stills from real-world executions for building the Tower, Twisted Tower, Triple-Tower and 3-π structures are shown in Fig. 6.3.



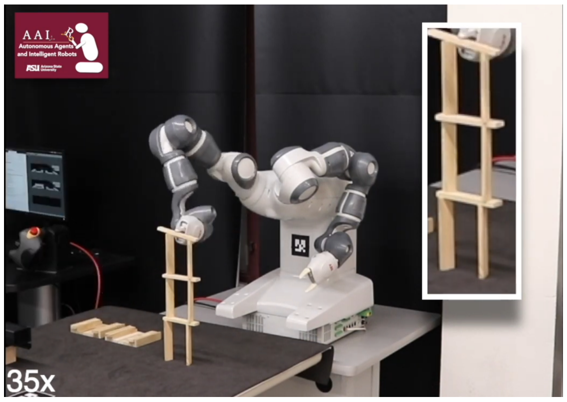
(a) Tower



(b) Twisted Tower



(c) Triple Tower



(d) 3-π

Figure 6.3: Real-World Keva Executions With Yumi-IRB 14000

Chapter 7

CONCLUSIONS

In this thesis, a methodology to compute generalized partial policies is developed, and a quick but greedy and incomplete method is developed to use the formed generalized policy to find concrete partial policies for unseen problems, including those with greater object counts. The methodology developed to learn generalized plans uses canonical abstraction for state representation, which allows for translating concrete example policies into their abstract representations in which patterns such as loops that are agnostic to problem-specific properties are identified. Results indicate that the approach presented in this work can generalize well to problems instances with much larger object counts. The generalized policy formed is shown to be a succinct representation of the underlying abstract structures seen in examples. It is also shown that the generalized policy formed using the methodology presented in this work is unique, regardless of the order in which example policies were used to learn it. The applicability of this methodology is demonstrated using multiple discrete MDP benchmark domains. The performance of the greedy method developed is compared against state-of-the-art discrete MDP solvers, and results indicate that for the benchmark domains considered, the method presented can solve problems much faster than the benchmarks in most cases.

Finally, the method developed in this work is incorporated with a framework that performs task and motion planning in stochastic environments. Using this method to perform task planning in this framework shows a significant reduction in run time required to achieve the task of building a free-standing Keva tower structure with 12 levels when compared to using state-of-the-art MDP solvers to do the same.

Chapter 8

FUTURE WORK

The GPI algorithm can find solutions quickly but is still susceptible to Sussman anomaly. Encoding goal hints, as shown in (Karia and Srivastava (2020)), while learning the generalized policy could possibly help in alleviating this problem. There is scope for improvement in the methodology used to construct the generalized policies from example policies. The methodology described does require some domain analysis to be done to determine the set of training examples to be used for learning the generalized policy. This can be automated by using an approach similar to the one described in (Srivastava *et al.* (2011b)). The patterns found by the tracing approach are not fully utilized by the way generalized policies are formed in this work. There is scope for improvement in constructing better policies using these traces. The solutions found by GPI are sub-optimal. Future directions could try to incorporate generalized policies learnt with LAO* to find near-optimal solutions faster.

In this work, the Merge algorithm utilizes all example policies given to it. If bad example policies which are sub-optimal solutions to problems are used, they can significantly impact the performance of GPI. More work can be done in identifying bad example policies. There is scope for improvement in the heuristic function used by GPI.

Loops are inherently unsafe structures. Some of the loops formed by the methodology developed in this work could result in situations where if the state-space for solving the MDP were infinite, GPI would never terminate. Further research to form generalized policies with termination guarantees can be done .

REFERENCES

Beeson, P. and B. Ames, “Trac-ik: An open-source library for improved solving of generic inverse kinematics”, in “2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)”, pp. 928–935 (2015).

Belle, V. and H. Levesque, “Foundations for generalized planning in unbounded stochastic domains”, in “KR”, pp. 380–389 (2016).

Berenson, D., S. S. Srinivasa, D. Ferguson and J. J. Kuffner, “Manipulation planning on constraint manifolds”, in “ICRA”, pp. 625–632 (IEEE, 2009).

Bertsekas, D. P. and J. N. Tsitsiklis, “An analysis of stochastic shortest path problems”, Mathematics of Operations Research **16**, 3, 580–595 (1991).

Bonet, B. and H. Geffner, “Labeled RTDP: improving the convergence of real-time dynamic programming”, in “ICAPS”, pp. 12–21 (2003).

Bonet, B., H. Palacios and H. Geffner, “Automatic derivation of memoryless policies and finite-state controllers using classical planners.”, in “ICAPS”, (2009).

Boutilier, C., R. Reiter and B. Price, “Symbolic dynamic programming for first-order mdps”, in “IJCAI”, vol. 1, pp. 690–700 (2001).

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

Diankov, R. and J. Kuffner, “Openrave: A planning architecture for autonomous robotics”, Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34 **79** (2008).

Fikes, R. E. and N. J. Nilsson, “Strips: A new approach to the application of theorem proving to problem solving”, Artificial intelligence **2**, 3-4, 189–208 (1971).

Flouri, T., K. Kobert, S. P. Pissis and A. Stamatakis, “An optimal algorithm for computing all subtree repeats in trees”, in “IWOCA”, vol. 8288, pp. 269–282 (2013).

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

Hoffmann, J., “A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm”, in “ISMIS”, pp. 216–227 (2000).

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

Karia, R. and S. Srivastava, “Learning generalized relational heuristic networks for model-agnostic planning”, arXiv preprint arXiv:2007.06702 (2020).

Kolobov, A., Mausam and D. S. Weld, “A theory of goal-oriented mdps with dead ends”, in “UAI”, pp. 438–447 (2012).

Koval, M., “Prpy”, URL `https://github.com/personalrobotics/prpy/` (2014).

Kumar, K., *Hierarchical Manipulation for Constructing Free Standing Structures*, Master’s thesis, Arizona State University (2019).

Mahler, J. and J. Liang, “Yumipy”, URL `https://github.com/BerkeleyAutomation/yumipy/` (2017).

McDermott, D., M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld and D. Wilkins, “Pddl-the planning domain definition language”, AIPS-98 planning committee **3**, 14 (1998).

Palacios, H. and H. Geffner, “From conformant into classical planning: Efficient translations that may be complete too.”, in “ICAPS”, pp. 264–271 (2007).

Peot, M. A. and D. E. Smith, “Conditional nonlinear planning”, in “AIPS”, pp. 189–197 (Elsevier, 1992).

Pineda, L., “Mdp-lib”, URL `https://github.com/luisenp/mdp-lib/` (2014).

Sagiv, M., T. Reps and R. Wilhelm, “Parametric shape analysis via 3-valued logic”, ACM Transactions on Programming Languages and Systems **24**, 3, 217–298 (2002).

Sanner, S. and C. Boutilier, “Practical solution techniques for first-order mdps”, Artificial Intelligence **173**, 5-6, 748–788 (2009).

Segovia-Aguas, J., S. Jiménez and A. Jonsson, “Computing programs for generalized planning using a classical planner”, Artificial Intelligence **272**, 52–85 (2019).

Segovia-Aguas, J., S. Jiménez and A. Jonsson, “Generalized planning with positive and negative examples”, in “AAAI”, vol. 34, pp. 9949–9956 (2020).

Shah, N., D. K. Vasudevan, K. Kumar, P. Kamojjhala and S. Srivastava, “Anytime integrated task and motion policies for stochastic environments”, in “ICRA”, pp. 9285–9291 (2020).

Srivastava, S., N. Immerman and S. Zilberstein, “Learning generalized plans using abstract counting.”, in “AAAI”, vol. 8, pp. 991–997 (2008).

Srivastava, S., N. Immerman and S. Zilberstein, “Merging example plans into generalized plans for non-deterministic environments”, in “AAMAS”, (2010).

Srivastava, S., N. Immerman and S. Zilberstein, “A new representation and associated algorithms for generalized planning”, Artificial Intelligence **175**, 2, 615–647 (2011a).

Srivastava, S., N. Immerman, S. Zilberstein and T. Zhang, “Directed search for generalized plans using classical planners”, in “ICAPS”, (2011b).

Steinmetz, M., J. Hoffmann and O. Buffet, “Revisiting goal probability analysis in probabilistic planning”, in “ICAPS”, (2016).

Winner, E. and M. Veloso, “Loopdistill: Learning looping domain-specific planners from example plans”, in “ICAPS”, (2007).

Winner, E. and M. M. Veloso, “Distill: Learning domain-specific planners by example”, in “ICML”, pp. 800–807 (2003).

Younes, H. L. and M. L. Littman, “Ppddl 1.0: An extension to pddl for expressing planning domains with probabilistic effects”, Techn. Rep. CMU-CS-04-162 **2**, 99 (2004).

APPENDIX A

NOTES ON CHAPTER 6

Chapter 6 of this document contains images from (Shah *et al.* (2020)) which was published in the ICRA 2020 conference. I was a co-author of this paper along with Naman Shah, Kislay Kumar, Pranav Kamojjhala and Siddharth Srivastava. The co-authors approve of the inclusion of the images in this thesis document.