Exploiting Relevance to Improve Robustness and Flexibility in Plan Generation and Execution

by

Christian Muise

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Exploiting Relevance to Improve Robustness and Flexibility in Plan Generation and Execution

Christian Muise
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2014

Automated plan generation and execution is an essential component of most autonomous agents. An agent's model of the world is often incomplete or incorrect, and its environment is typically noisy. To account for potential discrepancies between the agent's model of the world and the true state of the world, the planning techniques and representations used should enable flexible and robust agent behaviour. The agent should react swiftly when unexpected changes occur to assess the impact of the discrepancy and to accommodate as necessary. In particular, the agent should avoid unnecessary replanning and recognize changes that are irrelevant for its plan to achieve the goal.

In this dissertation we address various aspects of the planning process including (1) how to synthesize a plan, (2) what a plan should constitute and how we should represent one, and (3) how to effectively execute a plan. We enable robust and flexible agent behaviour by exploiting the notion of relevance in each of the key planning areas. Intuitively, relevance characterizes what is important to consider as a sufficient condition for some property to hold. We apply relevance to the key areas of automated planning to achieve the following contributions: (1) increased flexibility of partial-order plans, (2) improved robustness of partial-order plan execution, (3) robust execution of temporally constrained plans, and (4) improved efficiency of plan generation with non-deterministic action effects.

To increase the flexibility of partial-order plans, we introduce an effective method for generating optimally relaxed partial-order plans. For the execution of a partial-

order plan, we leverage regression to generalize an input plan, resulting in an execution monitoring framework that is far more robust than previous approaches. We incorporate the expressive power of temporal constraints and provide a means for monitoring the execution of a temporally constrained plan, building on our approach for executing a partial-order plan. Finally, we introduce a suite of techniques that leverage relevance to produce a state-of-the-art planner for domains with non-deterministic action effects. For each of these four areas, we investigate the theoretical properties surrounding our methods and empirically demonstrate their feasibility in comparison to the previous state of the art.

Dedication

For K.

Acknowledgements

I must first express my gratitude to my co-supervisors Sheila McIlraith and Chris Beck for the support and guidance they gave me during my graduate studies. The two have assessed countless bad ideas to stumble across those few of mine that worked out in the end, and both have endured far too many incoherent white-board explanations of mine to try and understand my point of view. Without their perfect blend of support and constructive criticism, I would have never achieved as much as I have. They have both been a wonderful mentor to me during my PhD and a source of inspiration.

I am grateful to the members my internal committee – Hector Levesque and Fahiem Bacchus – for the enlightening conversations, feedback, and ideas they have shared with me. They have helped shape my dissertation in an integral way. I also thank the members of the final thesis committee – Sylvie Thiebaux and Periklis Andritsos – for their insightful feedback and support.

I have had the privilege to share an office with many talented colleagues and friends. Eric Hsu was instrumental in throwing me into the deep end of research during my first month as a graduate student, and I have ridden that momentum ever since. Thanks go to all of the students in the KR group whom I've shared ideas with over the years, including Eric, Christian Fritz, Jorge Baier, Shirin Sohrabi, Vaishak Belle, Toby Hu, Marina Gueroussova, Letao Wang, Farah Juma, Sammy Davis-Mendelow, Brent Mombourquette, Toryn Klassen, Akshay Ganeshen, Daniel Kats, Torsten Hahmann, Jessica Davis, and Alexandra Goultiaeva. I am also very grateful for being welcomed into Chris Beck's group, where I had the continuous opportunity to expand my horizons to a range of wonderful research areas. Thanks go to everyone who I've had a chance to share ideas (and birthday cake) with, including Daria Terekhov, T.K. Feng, Tiago Stegun Vaquero, Tony Tran, Wen-Yang Ku, Maliheh Aramon, Maher Alhossaini, and Angela Glover.

Nathan Robinson has shared countless lunch and coffee breaks with me to push the boundaries of our academic imagination, and I owe a great deal to him for the insightful conversations we shared. The members of the Strathclyde Planning Group (now of King's College, London) have also been instrumental in my early development as a student. In particular, the guidance of Andrew and Amanda Coles helped propel me into the field of Automated Planning quicker than I could have ever hoped.

My family has unconditionally supported every interest of mine. I am eternally grateful to all of them for being so supportive in every challenge I've decided to take on.

Finally, none of this would have been remotely possible without the loving support of my wife, Kristie. She has carried me through the hard times, and I am extremely lucky to have shared all of the highs and lows together with her.

Contents

Chapter 1

Introduction

The study of how agents act in an environment is a cornerstone of artificial intelligence. To build an effective agent, one must consider a range of issues including (1) how the agent deliberates about what actions to perform (i.e., plan generation), (2) how the agent represents its strategy or plan for acting (i.e., plan representation), and (3) how the agent goes about executing its strategy (i.e., plan execution). In this dissertation we focus on techniques for each of these key issues with an aim to improve the overall flexibility and robustness of the acting agent.

When an agent takes action in the real world its model of the environment may be flawed or imprecise. We would like the agent to be as robust as possible in such scenarios; to adapt to unexpected changes in the environment, and to be flexible enough to react nimbly and alter the prescribed plan when it must. We can achieve this objective by exploiting the rationale of the planning process, when representing and executing the agent's plan. Our contribution to improving how an agent interacts with the world is situated in the field of *automated planning*.

In artificial intelligence, automated planning is the field of research concerned with achieving a goal specification through a synthesized plan composed of actions for an agent to perform. Typically, we describe a planning problem as an axiomatization of the actions that can be performed and their effect on the environment, as well as a specification of the initial state and the goal that the agent must achieve. Automated planning addresses the key aspects of agent execution that we are concerned with: plan generation, representation, and execution. The problem of plan generation is to synthesize a strategy for the agent to follow. Plan representation, on the other hand, refers to both *what* constitutes a plan (e.g., a policy mapping the state of the world to the appropriate action), as well as *how* the plan is represented (e.g., the data-structure used). Finally, the problem of plan execution refers to how the agent uses the representation of a plan

to effectively act in the environment.

There are a variety of assumptions that play a role in both the statement of the planning problem itself and in the plan representation. If the problem has non-deterministic action outcomes for example, the representation is better suited as a conditional plan or policy – ideally the plan should accommodate every potential outcome of an action that is chosen. In cases where the problem contains temporal constraints between actions, the plan representation must be able to express temporal relations between actions and action duration.

The diversity of tasks that can be reduced to planning problems makes automated planning an extremely powerful tool for many areas of artificial intelligence. As such, considerable research during the last few decades has focused on synthesizing and executing plans of various forms [42]. Central to much of the existing research are techniques for improving the *robustness* and *flexibility* of the planning process, and we define them both intuitively as follows.

> The *flexibility* of a plan or plan execution framework refers to the amount of discretion the executing agent has. When the number of possible realizations of a plan is higher than for another plan, we consider the former plan to be more flexible.
>
> *Robustness* is a measure of the agent's ability to achieve the goal over all possible scenarios during execution. We strive to empower the executing agent to be able to continue executing in as many situations as possible.

During plan generation and execution, we assume the agent has access to some model of what facts hold in the environment: the *state of the world*. We further assume that the agent has a model of the world's dynamics, including the actions available to the agent along with their effects on the state of the world. Often, however, the model of the world available to the agent is imperfect, which leads to changes in the state of the world that deviate from those predicted. To accommodate for these potentially imprecise models, a suitable planning process is vital: the agent's execution should be robust to minor discrepancies, the plan representation should be flexible to allow for different ways of executing, etc. Without flexibility and robustness, an agent executing a plan is susceptible to failure. The key insight we draw on to improve the flexibility and robustness in the overall planning process is *relevance*.

In practice, large portions of a planning problem may be irrelevant to the task of achieving the goal. For example, if a robot must drive from location A to location B,

the amount of fuel available is relevant while the colour of the fuel tank is not. Another form of relevance arises when considering a plan representation, as certain actions and ordering restrictions between the actions may be irrelevant to the validity of the plan. For example, getting money from an ATM before paying for lunch is a relevant ordering constraint between the two actions, but it may be irrelevant to order the action of calling your friend before or after going to the ATM.

A fundamental hypothesis of this dissertation is that effective plan generation and execution techniques require the consideration of the various forms of relevance that arise. Failure to do so leads to wasted effort and produces brittle solutions that are bound to fail in the presence of discrepancies in the state of the world. In this dissertation, we focus on identifying key forms of relevance that, when handled with care, allow for both theoretical and empirical improvements over the state-of-the-art planning techniques. Generally, we say that something is relevant if removing it precludes us from achieving some objective. In the above examples, the fuel in the gas tank and the ordering between getting money from the ATM and paying for lunch are both relevant for the objective of having a valid plan.

The techniques we introduce leverage three complementary notions of relevance – ordering, temporal, and state. While the abstract notion of relevance is applicable to a wide range of automated planning techniques, we focus on four primary areas in this dissertation:

1. Increasing the flexibility of partial-order plans
2. Improving the robustness of partial-order plan execution
3. Introducing a compelling framework for robustly executing plans that contain temporal constraints
4. Improving the robustness of plans for problems with non-deterministic action effects

By focusing on relevance, we improve the scalability of many planning techniques; many previously unsolvable problems are now readily handled. The improved robustness that our methods offer comes from both the plan generation techniques (i.e., generating a flexible partial-order plan) and plan execution techniques (i.e., generalizing a plan for robust execution). The theoretical and empirical outcomes of this dissertation improve both the flexibility and robustness of automated planning.

After formally presenting our thesis statement, we introduce the general techniques presented in this dissertation for improving plan generation, representation, and execu-

tion. This section provides the overview of how the various forms of relevance relate to one another, and how we employ them in solving the four key areas defined above.

Thesis Statement

Logical techniques for determining what is relevant to successfully generate and/or execute a plan can improve the robustness and flexibility of plan representation, generation, and execution. Methods such as logical regression provide a means to compute what is relevant in the state of the world, while methods from temporal dispatching provide a characterization of what is temporally relevant for a scheduled plan to be successful. Incorporating a rich notion of relevance into plan generation and execution techniques improves scalability and enables an autonomous agent to react robustly in a dynamic or uncertain environment.

1.1 Approach

To improve the flexibility and robustness of plan generation and execution techniques, we appeal to a common approach: leveraging *relevance*. The prevailing theme of our work is to develop theoretical frameworks and efficient implementations that focus only on what is relevant to achieve the task at hand; be it the relevant portion of state to achieve the goal with a plan fragment or the relevant portion of state for no plan to exist. With various forms of relevance in mind, we both augment existing techniques to improve their efficiency and introduce new methods for plan representation, generation, and execution.

The forms of relevance that we consider are *state* relevance, *ordering* relevance, and *temporal* relevance. State relevance (cf. Section 4.3) refers to focusing on a subset of the state of the world; commonly referred to as a *partial state*. A partial state compactly represents many states of the world, and we strive to determine theoretical properties that simultaneously hold for all states represented by a partial state. We use temporal relevance (cf. Section 5.6) to refer to the subset of an agent's execution history that is pertinent to the continued satisfaction of a set of temporal constraints. As we see in Chapter 5, reasoning about how an agent can satisfy temporal constraints is greatly simplified when it can be proved that only a relatively small subset of the history must be considered. Finally, ordering relevance (cf. Section 3.3) refers to the ordering constraints in a plan that are required for the plan's validity. By keeping only the relevant ordering constraints, we can maximize the flexibility of the resulting plan.

Focusing on relevance to improve automated planning is not a new concept; historically, various forms of relevance have been used successfully. Both state relevance and ordering relevance are central components in partial-order causal link (POCL) plan generation techniques [99]. Using state relevance is also at the core of the robust execution of sequential plans, starting with the triangle tables in the PLANEX system for the robot Shakey [34] and more recently with the continuous monitoring of plan optimality (e.g., the work of Fritz [37]). Relevance has also been leveraged for determining which operators should be considered for effectively solving a classical planning problem [47]. Temporal relevance plays an important role in the continuous monitoring of temporally extended goals and temporal monitors in systems such as HPlan-P [6] and LTL_3 Tools [7]. More generally, relevance is further considered in the AI literature in areas such as conditional independence [66], feature subset selection [8], and so on. In Section 2.3 we take a look at the related literature in further detail.

In this section we introduce each aspect of plan generation and execution that we tackle through the use of relevance.

Optimally Relaxing Plans

A plan typically consists of a set of actions and a set of ordering constraints over those actions. If the actions are totally ordered, then we say that the plan is *sequential*. As a generalization, a *partial-order plan* (POP) may contain actions that are unordered [99]. A *linearization* is a total ordering of the actions in a plan that respects the plan's ordering constraints, and in a *valid* POP every linearization achieves the goal from the initial state.

A POP is a compact representation of many linearizations that share the same actions. As such, an agent has far more flexibility when executing a POP as opposed to a sequential plan. It is this inherent flexibility that is appealing to us, as it directly correlates with the robustness of plan execution: having more options when executing allows an agent to delay committing to some decisions until absolutely necessary. There are two prevailing methods for producing a POP: (1) synthesize it from scratch (e.g., VHPOP [104] and POPF [18]) or (2) generate a sequential plan and convert it to a POP (e.g., the approaches by Kambhampati and Kedar [53] and Bäckström [4]). The latter approach is referred to as *partial-order relaxation*.

Unfortunately, traditional techniques for generating a POP from scratch are inadequate for many problems of reasonable size (see, for example, the International Planning Competition [40]). While forward-search state-based planners have improved greatly over the last decade, the partial-order planners have not followed suit. Further, from a theo-

retical standpoint, partial-order relaxation is known to be hard as well: it is NP-Hard to optimally relax a sequential plan [4].

To address this limitation, we turn to a field that has seen large improvements over the last decade: the study of *maximum satisfiability* (MaxSAT). In particular, we develop a novel partial-weighted MaxSAT encoding based on the ordering constraints relevant to the partial-order plan's validity. Every solution of the resulting encoding corresponds to an optimal partial-order relaxation of the input plan. Despite the theoretical difficulty, we are readily able to compute optimally relaxed partial-order plans that minimize not only the number of ordering constraints required (thus maximizing flexibility of the plan), but also minimize the number of actions in the resulting plan. Our use of ordering relevance, inspired by the historical methods for computing a POP, allows us to take advantage of state-of-the-art MaxSAT solvers.

Executing Partial-Order Plans

The typical approach to executing a POP is for the agent to execute the actions one after another while respecting the ordering constraints in the POP. The agent can proceed to do so as long as the next action it selects is applicable in the state of the world. When more than one unexecuted action appears at the start of a POP, the agent can arbitrarily pick the one that is most appropriate at run-time. This method provides flexibility during execution, as the agent can delay some of the ordering choices until run-time.

A more advanced execution strategy for a sequential plan is to employ a form of *execution monitoring* pioneered by Fritz and McIlraith in a variety of papers including [38] and most thoroughly summarized in Fritz's thesis [37]: at every step while executing, the agent (1) evaluates the state, (2) determines if some fragment of the plan is applicable, and if so (3) determines the most appropriate next action. Fritz expanded this notion to consider the conditions for plan optimality as well as the conditions for near-optimal policies in stochastic domains. In the case of sequential plans, the existing technique of Fritz performs robust execution monitoring by generalizing the plan to a policy with the following behaviour: if any suffix of the sequential plan can achieve the goal in the current state, execute the action at the start of the smallest such suffix.

In our work, we aim to combine the flexibility of a POP with the robustness of execution monitoring. Limiting the execution monitoring techniques to a single sequential plan is quite restrictive, as the agent only has a small number of plan fragments at its disposal to achieve the goal (n suffixes if n is the number of actions in the plan). In contrast, forcing every action to occur once and only once, as is the case with typical POP execution, is overly restrictive in a dynamic environment.

To accomplish the task of robust execution monitoring using a POP, we appeal to state relevance in determining the conditions under which a partial plan fragment can achieve the goal. These conditions are computed by using logical regression of the goal through the partial plan fragment. We realize this through a systematic process that continuously partitions the POP and implicitly enumerates the $k \cdot n$ partial plan sequences available (where n is the number of actions and k is the number of linearizations).

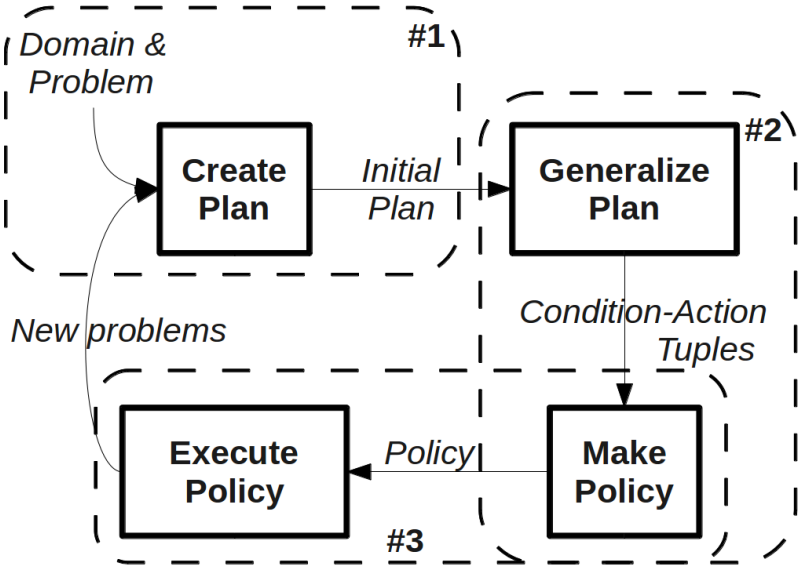


Figure 1.1: Overview of the approach for execution monitoring of a POP. Phase #1 creates a POP to be executed. Phase #2 generalizes the POP into a list of condition-action tuples and subsequently a policy. Phase #3 executes the policy online.

Figure 1.1 shows an overview of our approach. In the first phase, we compute a POP using a relaxation of a sequential plan (cf. Chapter 3). The second phase of our approach is the plan generalization. We first systematically compute an ordered list of condition-action tuples, where the order corresponds to the required cost to achieve the goal. The "condition" in every tuple is a partial state, and the agent will execute the action from the first tuple in the list where the condition holds in the current state. That expected behaviour is embodied in a policy for efficient lookup, which is then subsequently fed to the agent for the third phase: online execution. During this phase, the agent consults the policy at every state and executes the returned action. If the policy does not return an action, then no suffix of any linearization can achieve the goal and replanning is necessary.

Executing Temporally Constrained Plans

Complementary to the study of executing plans, the area of schedule dispatching involves the task of assigning a time to every element in a set of events while satisfying a set of temporal constraints. Unlike planning, there is no notion of what is true or false in the environment. The typical execution strategy is similar to that of traditional POP execution strategies – execute the events one after another until they have all occurred.

Many attempts to integrate the fields of automated planning and schedule dispatching come with the assumption that the agent will execute every action or event exactly once. This largely restricts the acting agent in the same manner as traditional POP execution. The drawback of considering just the scheduling aspect of a problem is that the environment, potentially dynamic in nature, is ignored during execution. Conversely, the execution of a POP typically lacks the expressiveness required for specifying temporal constraints between action executions.

To overcome these two limitations, we introduce a formal framework for specifying a temporally constrained partial-order plan (TPOP). Following our methods for the robust execution of a POP, we do not want to limit the execution of a TPOP by having every action executed exactly once. Allowing multiple action executions, however, necessitates a formal interpretation of temporal constraint satisfaction. We present a set of temporal constraint types, along with their semantic interpretations, to handle the possibility of actions appearing zero, one, or multiple times.

We expand the techniques of our POP execution to include temporal reasoning. When considering if a partial plan can achieve the goal, we further analyze the temporal constraints and compute a template for the *relevant* part of execution history. The template is a form of temporal relevance that allows the agent to restrict the reasoning online to a very limited subset of what has already occurred. The temporal reasoning techniques we include are a suite of algorithms from the scheduling community that allow us to determine precisely when a partial plan will be executable according to the temporal constraints.

Generating Non-deterministic Plans

In classical planning, the outcome of an action is fully specified – the agent knows precisely what state will be reached after it executes an action. In *non-deterministic* planning, the outcome of the action is uncertain at planning time, but known after execution of the action. Ideally, a planning procedure should consider every one of the (finite) possible outcomes of an action. The desired plan, therefore, is not simply a set of actions and

ordering constraints; it is a policy that maps the state of the world to the appropriate action. We are primarily concerned with producing *strong cyclic solutions*, which are policies that are guaranteed to eventually achieve the goal and permit the agent to repeatedly visit the same state.[1] A *strong plan* is a policy that guarantees the goal is reached in a finite number of steps, thus forbidding the possibility of reaching the same state more than once. Another important type of plan in the non-deterministic setting is a *weak plan*: a plan that has a non-zero chance of reaching the goal.

Previous state-of-the-art techniques for producing strong cyclic plans (e.g., NDP [64] and FIP [39]) involve simulating all of the possible outcomes and exhaustively enumerating the states reachable by the constructed policy. When the simulation reaches a state that the policy does not know how to handle, the algorithm computes a new weak plan and updates the policy accordingly. There are a number of problems with the previous techniques; the most drastic being that the policy for the strong cyclic plan is stored as a mapping of complete states to actions. Other issues include the exhaustive enumeration of the reachable states, poor handling of dead-end states that may be present, and the unnecessary use of complete states in many parts of the policy repair process.

By leveraging state relevance, we improve on many aspects of the previous non-deterministic planning techniques. The policies our method generates are exponentially smaller due to the fact that mapping a partial state to an action is equivalent to succinctly representing the mapping of a large number of states to an action. When we find a dead-end state (i.e., a state in which no solution exists), we generalize it by discarding all fluents that are irrelevant with respect to the realization of the dead-end property. Doing so provides us with a characterization of a large number of states that should be avoided. We also see significant improvement in the efficiency of our planner due to the modification of the simulation phase. Rather than exhaustively enumerating the reachable state space of the current policy, we determine relevant stopping conditions based on the state of the world. Whenever the relevant condition (i.e., a partial state) holds in the simulation, we know the policy is a strong cyclic solution from that point on, and the simulation need not expand further.

1.2 Contributions

Broadly, the contributions of this dissertation are as follows:

[1] For strong cyclic solutions there is a fairness assumption that stipulates every action outcome will occur infinitely often if the action is executed infinitely often.

Optimally Relaxing Partial-Order Plans (Chapter 3)

In Chapter 3 we present a method for computing an optimally relaxed partial-order plan. This work appeared in both the COPLAS'11 workshop [75] and ICAPS'12 conference [77]. Our solution uses a novel partial-weighted MaxSAT encoding where solutions correspond to maximally flexible partial-order plans. We further introduce the notion of a *minimum cost least commitment POP*, characterized by having minimum total action cost and subsequently as few ordering constraints as possible. We demonstrate that an existing heuristic approach for relaxing plans is empirically extremely adept at computing a particular type of optimal relaxation, despite the algorithm's poor theoretical guarantee. We additionally demonstrate the feasibility of our approach compared to previous related work and provide theoretical results that characterize the plans we produce.

Robustly Executing Partial-Order Plans (Chapter 4)

In Chapter 4 we present a method for generalizing a partial-order plan (POP) for robust and efficient execution by an autonomous agent, originally published in IJCAI'11 [74]. We introduce a theoretical characterization of POP viability that allows us to implicitly enumerate every plan fragment of the POP and compute the sufficient condition for the fragment to achieve the goal. We present a method for compactly representing these conditions as a policy, and identify the characteristics of a POP that maximize flexibility. The resulting system is far more efficient and robust than previous methods. We measure efficiency by the time required for the agent to react, and we measure robustness by analytically computing the total number of states handled by the generalized plan. We further provide a theoretical basis for our approach, which lays the foundation for characterizing the notion of regression through a POP.

Robustly Executing Temporally Constrained Plans (Chapter 5)

In Chapter 5 we introduce a compelling framework and semantics for describing temporal constraints over actions that may be repeated multiple times. The work for this chapter appeared in IJCAI'13 [73]. We further present a means for robustly executing temporally constrained plans that builds on insights from the work in Chapter 4. To accomplish the necessary temporal reasoning, we appeal to a variety of schedule dispatching techniques. By careful analysis in an offline phase, we are able to restrict the reasoning during execution to a small relevant subset of a potentially

large execution history. The resulting system is able to dynamically reason about both logical and temporal requirements for a plan fragment to achieve the goal. The work serves as a unifying approach to both the execution of a POP and the dispatching of a temporal network. We provide both an implementation of the proposed system, and prove the desired properties of the approach.

Efficient Non-deterministic Planning (Chapter 6)

In Chapter 6 we introduce a state-of-the-art planner for solving fully observable non-deterministic (FOND) planning problems, published in ICAPS'12 [76]. We present a suite of principled methods to leverage the relevant part of the state of the world. These methods include (1) targeted replanning when the incumbent solution is insufficient, (2) compact and robust representation of the solution, (3) efficient dead-end detection and avoidance, and (4) powerful stopping conditions for the algorithm's simulation loop. Our system is able to compute robust solutions exponentially faster than previous methods, and the resulting policies are also exponentially smaller. The contributions represent a significant improvement in the scalability of non-deterministic planning and introduce the possibility for better probabilistic planning in domains that are difficult due to avoidable dead-ends.

1.3 Organization of Dissertation

In Chapter 2 we provide the necessary background and notation we use throughout the dissertation and further provide connections of our overall approach to existing literature. In Chapter 3 we present our contribution to the problem of relaxing partial-order plans to maximize flexibility. In Chapter 4 we describe our work on improving the robustness of partial-order plan execution. In Chapter 5 we describe the formalism for temporally constrained partial-order plans, and our approach to executing them. In Chapter 6 we detail our contributions to improving non-deterministic planning. Finally, we conclude in Chapter 7 and discuss possible future research directions.

Chapter 2

Preliminaries

This chapter describes the necessary background and notation required for the majority of the dissertation. We first describe the classical planning formalism, and then follow with a description of the various forms of plan representation that we will consider. We finish with a general overview of existing literature that relates to relevance in planning. The majority of this chapter is review of existing literature, with the exception of the Match Tree formalism. We have generalized the existing Match Tree formalism found within the literature for the purposes of this dissertation.

2.1 Classical Planning

In general, planning is the task of synthesizing a solution that dictates what actions an agent must take in order to achieve some prescribed goal. In classical planning, we assume the world is fully known and deterministic [89]. Classical planning has many applications that range from robotics to modelling biological processes [42]. The standard approach for synthesizing a classical plan is to perform search through the state space of a problem, using heuristics to guide the planner towards a high quality solution. Here, we describe the most common formalism used for specifying a planning problem.

Unless otherwise stated, we restrict ourselves to STRIPS planning problems [34]. In STRIPS, a planning problem is a tuple $\Pi = \langle F, O, I, G \rangle$ where F is a finite set of fluents, O is the finite set of operators, $I \subseteq F$ is the initial state, and $G \subseteq F$ is the goal state. We refer to a *complete state* (or just *state*) as a subset of F with the assumption that fluents that do not appear in a complete state are false in that state. A *partial state* is a subset of F that compactly represents the set of complete states that are a superset of the partial state. Fluents that do not appear in a partial state are assumed to be undefined. Because both complete and partial states are represented using a subset of F,

we will be explicit about what type of state is being referred to when the interpretation is ambiguous. We denote the set of all possible complete states for a planning problem as $\mathcal{S}$. In STRIPS, I is a complete state while G is a partial state.

We say that a state s *entails* the formula ψ, denoted as $s \models \psi$, if the conjunction of fluents in s with the negation of the fluents not in s logically entails the formula ψ. If ψ is a conjunction of positive fluents then $s \models \psi$ iff the conjuncts of ψ form a subset of s. Depending on the situation, we will treat a partial or complete state as either a set of fluents or conjunction of fluents.

We characterize an operator $o \in O$ by three sets:

- $PRE(o)$: The fluents that must be true in order for o to be executable.
- $ADD(o)$: The fluents that operator o adds to the state.
- $DEL(o)$: The fluents that operator o deletes from the state.

An *action* refers to a specific instance of an operator, and it inherits the PRE, ADD, and DEL sets of the matching operator. We say that an action a is *executable* in state s iff $PRE(a) \subseteq s$. The resulting state after executing action a in state s is defined as:

$$\mathcal{P}(s,a) \stackrel{\text{def}}{=} \begin{cases} (s \setminus DEL(a)) \cup ADD(a) & \text{if } a \text{ is executable in } s \\ \textit{undefined} & \text{otherwise} \end{cases}$$

For a planning problem $\Pi = \langle F, O, I, G \rangle$, we associate a cost function c_Π that maps every operator $o \in O$ to a non-negative real number (i.e., $c_\Pi : O \to \mathbb{R}_0^+$). We define the cost of an action similarly, each action inheriting the cost of its matching operator.

We will make use of two further items of notation with respect to a set of actions A:

- **adders**(f): The set of actions in A that add the fluent f:

$$\{a \mid a \in A \text{ and } f \in ADD(a)\}$$

- **deleters**(f): The set of actions in A that delete the fluent f:

$$\{a \mid a \in A \text{ and } f \in DEL(a)\}$$

2.2 Plan Representations

In this section we present plan representations that fall under two primary categories: prescribed plans and policies. A prescribed plan contains a set of actions that must be

executed and some ordering over them, while a policy simply maps the state of the world to the action that should be executed (or a predefined null value if the policy is partial).

The prescribed plan representations that we make use of include sequential plans and partial-order plans. In the case of non-uniform action costs, the plan representations do not change. We may, however, prefer plans that have a lower overall cost, which we measure as the sum of the costs for every action in the plan.

The policy representations that we make use of include ordered decision diagrams [5] and match trees [48]. Current literature contains many uses of the former to represent a planning policy (e.g., the work of Boutilier *et al.* [14], Younes *et al.* [105], and Fu *et al.* [39]), but to the best of our knowledge we are the first to use a match tree for representing a policy in planning.

Unless otherwise specified, the formalisms presented in this section follow the classical definitions in automated planning (see for example [89] and [42]).

2.2.1 Sequential Plans

The most common representation of a solution to a planning problem is a sequential plan. A sequence of actions $\vec{a} = [a_1, \cdots, a_n]$ is *executable* if the preconditions of each action in the sequence are true in the corresponding state, and an executable sequence of actions is a *sequential plan* for the problem $\Pi = \langle F, O, I, G \rangle$ if executing the actions in $\vec{a}$ in sequence, when starting in state I, causes the goal to hold in the final state:

$$\mathcal{P}(\mathcal{P}(\cdots \mathcal{P}(I, a_1) \cdots, a_{n-1}), a_n) \models G$$

For readability, we abbreviate the progression of a sequential plan $\vec{a}$ from state s as $\mathcal{P}^*(s, \vec{a})$. We refer to the *suffix* of a sequential plan (or sequence of actions) $\vec{a} = [a_1, \ldots, a_n]$ to be the empty sequence or the sequence of actions $[a_i, \ldots, a_n]$ where $i \geq 1$. We define the *prefix* of a sequential plan analogously. Finally, the cost of an action sequence $\vec{a} = [a_1, \ldots, a_n]$ is the sum of the individual actions costs:

$$c_\Pi(\vec{a}) = \sum_{i=1}^{n} c_\Pi(a_i)$$

2.2.2 Partial-order Plans

Rather than impose a total order on the actions in a plan, a partial-order plan (POP) specifies a set of ordering constraints over the actions. We define a POP with respect to a planning problem Π as a tuple $\langle \mathcal{A}, \mathcal{O} \rangle$ where $\mathcal{A}$ is the set of actions in the plan and

$\mathcal{O}$ is a set of ordering constraints between the actions in $\mathcal{A}$ [89]. While the same ground action may appear more than once in $\mathcal{A}$, we assume that every element of $\mathcal{A}$ is uniquely identifiable. For the actions $a_1, a_2 \in \mathcal{A}$, we denote the ordering constraint between a_1 and a_2 as $(a_1 \prec a_2) \in \mathcal{O}$ and interpret the constraint as "action a_1 appears before action a_2 in the plan". A total ordering of the actions in $\mathcal{A}$ that respects $\mathcal{O}$ is a *linearization*. A POP provides a compact representation for multiple linearizations.

To simplify the exposition in this dissertation, we will place a few assumptions on all of the POPs we consider, unless otherwise specified. First, the ordering constraints $\mathcal{O}$ are assumed to be transitively closed:

$$\forall a_1, a_2, a_3 \in \mathcal{A}, (a_1 \prec a_2) \wedge (a_2 \prec a_3) \rightarrow (a_1 \prec a_3)$$

Assuming that $\mathcal{O}$ is transitively closed does not change the fundamental structure of the POP – the set of linearizations remains the same – but it allows us to effectively compare the flexibility of two POPs that share the same action set.

The second assumption we will make is that unless otherwise specified, there are two specially designated actions in the POP that represent the initial state and goal state: a_I and a_G respectively. a_I is ordered before every other action, and a_G is analogously ordered after every other action. For a planning problem $\Pi = \langle F, O, I, G \rangle$, the actions have the following interpretation:

$$\begin{aligned} PRE(a_I) &= \emptyset & PRE(a_G) &= G \\ ADD(a_I) &= I & ADD(a_G) &= \emptyset \\ DEL(a_I) &= \emptyset & DEL(a_G) &= \emptyset \end{aligned}$$

The inclusion of a_I and a_G actions allow us to simplify the presentation of many algorithms, avoiding special checks in the procedure (e.g., we can assume that there will always be a "first" and "last" action in the POP).

Similar to the cost of an action sequence, the cost of a POP $P = \langle \mathcal{A}, \mathcal{O} \rangle$ is the sum of the action costs for the actions in P:

$$c_\Pi(P) = \sum_{a \in \mathcal{A}} c_\Pi(a)$$

Execution of a POP is also quite similar to that of a sequential plan: every action in the POP is executed one after the other as long as the ordering constraints are satisfied. Any complete execution of a POP will correspond to a linearization of the POP.

Algorithm 1 outlines the basic POP execution strategy.

Algorithm 1: POP Execution Algorithm

```
   Input: POP ⟨A, O⟩
1  while |A| > 0 do
       /* Candidates is set to every action at the start of the POP.   */
2      Candidates = {a | a ∈ A and ∀a' ∈ A, (a' ≺ a) ∉ O};
       /* We arbitrarily pick any one of the candidates.               */
3      a = PICK(Candidates);
       /* The action is executed and removed from the POP.             */
4      EXECUTE(a);
5      A = A − {a};
```

Depending on how the POP was constructed, it may include a set of causal links, $\mathcal{C}$. Each causal link contains a pair of ordered actions, a_1, a_2, and a fluent, p, such that the a_1 achieves p for a_2: denoted as $(a_1 \xrightarrow{p} a_2)$. Causal links often serve as justifications for the ordering constraints in a POP. For the majority of our work we do not exploit causal links, but they are useful in formalizing one notion of POP validity.

A POP P is valid for a planning problem Π if and only if every linearization of P is a sequential plan for Π (notion 1).[1] While simple and intuitive, notion 1 is rarely used to verify the validity of a POP because there may be a prohibitively large number of linearizations represented by the POP. There is, however, a tractable equivalent notion of POP validity that uses the concepts of causal links, open preconditions and threats.

For a POP $\langle \mathcal{A}, \mathcal{O} \rangle$ and a set of causal links $\mathcal{C}$, an *open precondition* is a precondition p of an action $a \in \mathcal{A}$ that does not have an associated causal link:

$$\nexists a' \in \mathcal{A} \text{ s.t. } (a' \xrightarrow{p} a) \in \mathcal{C}$$

If a precondition is not open, we say that it is *supported*, and we refer to the associated action in the causal link as the *achiever* for the precondition.

A *threat* in a POP refers to an action that can invalidate a causal link due to ordering constraints (or lack thereof). Formally, if $(a_1 \xrightarrow{p} a_2) \in \mathcal{C}$, we say that the action a_3 threatens the causal link $(a_1 \xrightarrow{p} a_2)$ if the following two conditions hold:

- We can order a_3 between a_1 and a_2:

$$\{(a_3 \prec a_1), (a_2 \prec a_3)\} \cap \mathcal{O} = \emptyset$$

[1] Note that notion 1 does not rely on the set of causal links $\mathcal{C}$.

- The action a_3 deletes p:

$$p \in DEL(a_3)$$

The existence of a threat means that a linearization may exist that is not executable. With the actions a_I and a_G included, we have the following theorem that characterizes the second notion of POP validity:

Theorem 1 (POP Validity [89]). *Given a planning problem* Π*, POP* $P = \langle \mathcal{A}, \mathcal{O} \rangle$ *and set of causal links* $\mathcal{C}$*,* P *is a valid POP for the planning problem* Π *if no action in* $\mathcal{A}$ *has an open precondition and no causal link in the set* $\mathcal{C}$ *has a threatening action in* $\mathcal{A}$*.*

The two notions of POP validity are similar in the sense that if the second notion holds, then the first follows. If the first notion holds for $\mathcal{A}$ and $\mathcal{O}$, then a set of causal links $\mathcal{C}$ exists such that the second notion holds for $\langle \mathcal{A}, \mathcal{O} \rangle$ and $\mathcal{C}$.

The final POP terminology that we will use is a *POP suffix*. A POP suffix of a given POP $\langle \mathcal{A}, \mathcal{O} \rangle$ is any POP $\langle \mathcal{A}', \mathcal{O}' \rangle$ where the following holds:

1. The suffix contains a subset of the actions:

$$\mathcal{A}' \subseteq \mathcal{A}$$

2. The ordering constraints in the suffix are induced by the actions:

$$\mathcal{O}' = \{(a_1 \prec a_2) \mid (a_1 \prec a_2) \in \mathcal{O} \text{ and } a_1, a_2 \in \mathcal{A}'\}$$

3. No action in the suffix is ordered before an action outside of the suffix:

$$\forall a_1 \in \mathcal{A}', ((a_1 \prec a_2) \in \mathcal{O}) \rightarrow (a_2 \in \mathcal{A}')$$

2.2.3 Policies and Ordered Decision Diagrams

Rather than specify the precise set of actions to be executed in a plan, one can represent a solution as a mapping from the state of the world to an appropriate action for execution. This mapping, referred to as a policy, is the prevailing representation used in problems where uncertainty is present (e.g., with non-deterministic action outcomes [17]). An Ordered Decision Diagram is a policy representation with a suite of appealing computational properties [24]. Historically, decision diagrams have been used in a range

of applications, but for this dissertation we are concerned with how they can be used to represent a policy that encompasses execution behaviour.

A *policy* is a function that maps states to actions [14]: $\mathcal{S} \rightarrow \mathcal{A}$. A *partial policy* is a function that maps a state to either an action or a designated undefined value: $\mathcal{S} \rightarrow \mathcal{A} \cup \{\perp\}$. In Chapter 4, we use an Ordered Algebraic Decision Diagram (OADD) [5] to represent a policy. An OADD is a rooted directed acyclic graph with a distinction made between the set of inner nodes and set of leaf nodes. We associate an action (or $\perp$) to every leaf node and a fluent to every inner node. Inner nodes have precisely two outgoing edges – a true and false edge corresponding to the truth of the fluent in the state of the world.

OADDs have one further restriction: the order of fluents from any root to leaf path must follow a predefined order. The order ensures that if we check two fluents on a path from the root to a leaf node, we will always check them in the same order. Restricting ourselves to ordered decision diagrams allows us to efficiently perform operations on the representation (e.g., combining two separate OADDs).

An Ordered Binary Decision Diagram (OBDD) [10] is similar to an OADD, with the main difference being that we associate either true or false to a leaf node and not an action. An OBDD is therefore a succinct representation of a Boolean function that maps states of the world to true or false: $\mathcal{S} \rightarrow \{true, false\}$. Given an OADD or OBDD, Pol, we will use the typical function notation, $Pol(s)$, to signify the action, *true*, or *false* that is returned given the state s.

2.2.4 Match Trees

While they have existed in a limited form previously in the literature, this section introduces the notion of match trees formally. Similar to OADDs, a *match tree* is a decision diagram that provides a compact mapping between two domains. However, instead of mapping a Boolean setting to a single object, a match tree maps a multi-valued setting to a set of objects. The match tree data structure was introduced to check for action applicability in the Fast Downward planning system [48].[2] More recently, however, it has been used by Eyerich and Helmert to represent the conditions for a plan of bounded length to achieve the goal [33]. Here, we present a slight generalization of the match tree data structure that we use in multiple parts of our research (cf. Chapters 5 and 6).

Definition 1 (Match Tree)**.** Let V be a set of variables that have the finite domain of $D(v)$ for $v \in V$, and let O be an arbitrary domain of objects. We denote the domain of

[2]In the journal article, the match tree data structure is referred to as the "successor generator".

all settings to $V = \{v_0, \cdots, v_n\}$ as $\mathbb{V} = D(v_0) \times \cdots \times D(v_n)$. A *match tree* $T(f)$ is a compact representation of the partial function f that maps a complete setting of V to a subset of the objects in O:

$$f : \mathbb{V} \rightarrow 2^O$$

It is important to emphasize that f is defined over *complete settings* of the variables in V. However, to specify f we typically start with a mapping of partial settings to a set of objects. A partial setting p to the variables V is said to *agree with* a complete setting $s \in \mathbb{V}$, denoted as $p \sim s$, if the variables defined in p have the same value in both p and s. We use $vars(p)$ to denote the variables that have a value defined in the partial setting p. Typically, we assume that f is provided as a set of $\langle p, \sigma \rangle$ pairs, where p is a partial assignment to V and $\sigma \in O$. The interpretation of $\langle p, \sigma \rangle$ is that for any setting $s \in \mathbb{V}$ that agrees with the partial setting p, we have $\sigma \in f(s)$. Given a set of $\langle p, \sigma \rangle$ pairs, $Pairs$, we can fully define f as follows:

$$f(s) = \{\sigma \mid \langle p, \sigma \rangle \in Pairs,\ s \sim p\}$$

We represent a match tree as a rooted tree and distinguish between inner nodes (those that have one or more outgoing edges) and leaf nodes (those that have no outgoing edges). Every inner node n of a match tree is associated with a variable $var(n) \in V$ and has precisely $|D(var(n))| + 1$ outgoing edges – the children denoted as $n[d]$ for all $d \in D(var(n))$. The final outgoing edge for every inner node is $n[*]$, and it represents a special situation where the variable's value does not matter. Finally, associated with every leaf node n is a set of objects $objs(n) \subseteq O$.

Unlike OADDs, the order of variables in a match tree is not predefined, but we do require that for any root-to-leaf path, a variable appears at most once. Additionally, the leaf nodes form a partition of the objects in O. That is, every object from O is associated with *exactly* one leaf node.

Algorithm 2 describes the recursive method for how a match tree $T(f)$ can be read for a complete setting $s \in \mathbb{V}$ to compute the set of objects $f(s)$. The initial call is made with the match tree's root node. Note that we use union on line 4 because the condition in a pair used to construct the match tree is a partial state – the pairs that have $var(n)$ undefined will be found through the call to READMT$(n[*], s)$.

If we have a total ordering over the objects in O then we can use a match tree as a representation for a partial mapping to a single object: $g : \mathbb{V} \rightarrow O$. As such, for the

Algorithm 2: READMT(n, s): Recursive method to read a match tree.

Input: A node n of a match tree and complete setting $s \in \mathbb{V}$
Output: Set of objects

```
1 if n is leaf then
2 |  return objs(n);
3 else
4 |  return ReadMT(n[*], s) ∪ ReadMT(n[s[var(n)]], s);
```

total ordering, assume we have a scoring function that maps an object in O to a natural number: $score : O \to \mathbb{N}$. We can thus define the function g as follows:

$$g(s) = \arg\min_{\sigma \in f(s)} score(\sigma)$$

To construct a match tree so that it behaves as a proper representation for the function $f : \mathbb{V} \to 2^O$, we use Algorithm 3 given the set $Pairs$ of $\langle p, \sigma \rangle$ pairs and a set of seen variables $Seen$ (initialized to the empty set). As the recursion goes deeper, we add to the seen variables to avoid re-checking the same variable on a root-to-leaf path.

Algorithm 3: MAKEMT($Pairs$, $Seen$): Recursive method to build a match tree.

Input: Set of $\langle p, \sigma \rangle$ pairs $Pairs$, and set of variables $Seen$
Output: Root node of the new match tree

```
1 if ∀⟨p, σ⟩ ∈ Pairs, vars(p) ⊆ Seen then
2 |  return new LeafNode({σ | ⟨p, σ⟩ ∈ Pairs});
  /* Create the inner node                                              */
3 n = new InnerNode();
  /* Pick a variable that has not been seen yet                         */
4 var(n) = PickVariable(Pairs, V − Seen);
  /* Recursively call the method for the pairs that match a value of var(n) */
5 for d ∈ D(var(n)) do
6 |  n[d] = MakeMT({⟨p, σ⟩ | ⟨p, σ⟩ ∈ Pairs, p[var(n)] = d}, Seen ∪ {var(n)});
  /* Recursively call the method for the pairs that are not dependent on var(n) */
7 n[*] = MakeMT({⟨p, σ⟩ | ⟨p, σ⟩ ∈ Pairs, p[var(n)] = undefined}, Seen ∪ {var(n)});
  /* Return the final constructed match tree rooted at n               */
8 return n;
```

Note that as we enumerate over every possible value in the domain of $var(n)$, including *undefined*, we form a partition of the set of pairs $Pairs$ (lines 5-7). This ensures that an

object from O appears in exactly one leaf node. Central to the efficiency of the match tree is how we select a new branching variable on line 4. Later, we will describe the heuristic we use for our work, but for completeness we present the heuristic traditionally used by the Fast Downward (FD) planning system. The FD heuristic for building a match tree assumes that we have a total ordering of the variables in V.[3] For convenience, we assign a unique score, $score(v)$, to each variable $v \in V$. When PICKVARIABLE($Pairs$, $V - Seen$) is called, we select the variable with the smallest score that is mentioned in $Pairs$:

$$\text{PICKVARIABLE}(Pairs, V - Seen) = \min_{v \in (V - Seen) \cap PairVars} score(v)$$
$$\text{where,} \qquad PairVars = \bigcup_{\langle p,\sigma\rangle \in Pairs} vars(p)$$

Using $PairVars$ is a small optimization that avoids creating an inner node where every child but $n[*]$ is an empty leaf node. For its use in the FD system, the naïve heuristic does just fine [48], but as we shall see later, it can be substantially improved.

While match trees are extremely effective in practice, the computational complexity of using them to represent a function is not as appealing compared to OADD's. In particular, the READMT operation may need to explore the entire match tree even if the returned set of objects is empty. Nevertheless, match trees prove to be quite useful for the areas we employ them in our research.

2.3 Relevance in Planning

Relevance has appeared in the planning literature in multiple guises (e.g., the range of contributions to reasoning about action and change in the AAAI Symposium Series on Relevance [44]). Nebel *et al.* formalize notions of relevance to include the set of operators that are required to achieve the planning goal and the set of fluents that are required for the application of relevant operators [79]. They approximate their notions of relevance to improve the planning process by way of pruning out irrelevant operators and fluents from the domain in a preprocessing step that uses syntactic analysis of the actions in the domain. Their approximation, however, is aggressive in the selection of operators and fluents that are deemed irrelevant – the resulting planning problem may be unsolvable while the original problem is not.

More recently, Haslum *et al.* follow the same motivation to focus on determining

[3]The FD system selects this order based on statistics from the causal graph.

the operators that are irrelevant to a planning problem [47]. They identify irrelevant operators in unary planning domains (i.e., those where actions have only one effect) by using path-based relevance analysis: a technique that attempts to identify when a path in the state space is dominated by another with a smaller set of actions. Unlike the approach by Nebel *et al.* the approach by Haslum *et al.* is a sound procedure.

In the area of planning under uncertainty, relevance plays a central role in efficiently maintaining a compact representation of a belief state (i.e., a set of possible states the world may be in). Son and Tu partition the representation of the belief state into independent partial representations by grouping the literals in the domain based on those that are relevant to one another [94]. Similarly, Palacios and Geffner use a notion of relevance between literals to inform a process that compiles away uncertainty into a classical planning problem [80]. Here, two literals are considered relevant if the knowledge of one being true or false is required for the achievement of another. Following on the previous approaches that formalize relevance between a pair of literals in a domain with uncertainty, Bonet and Geffner extend the notion of relevance to the more general form of "causal relevance" [12]. Here, two literals are considered relevant if one causally implies the other, whereas previous notions additionally considered two literals to be relevant if one is related to the other through some observation.

More directly related to this dissertation are the concepts of *regression* and *execution monitoring*. In this section, we discuss each in further detail.

2.3.1 Regression

The notion of *regression* refers to a form of syntactic rewriting that allows us to compute the weakest condition that must hold prior to the execution of an action in order for a formula to hold after the action occurs. Introduced originally by Waldinger in 1977 [98], it was later exploited and refined by Reiter in the context of situation calculus [86]. For our purposes, we exploit a simple form of regression, restricted to STRIPS, and limit our exposition accordingly. We formally define regression as follows:

Definition 2 (Regression in STRIPS)**.** Given a planning problem Π and a conjunction of fluents, ψ, expressed as a set of fluents, we define the *regression* of a conjunctive formula ψ with respect to an action a, denoted $\mathcal{R}(\psi, a)$, as follows:

$$\mathcal{R}(\psi, a) = \begin{cases} (\psi \setminus ADD(a)) \cup PRE(a) & \text{if } DEL(a) \cap \psi = \emptyset \\ \textit{undefined} & \text{otherwise} \end{cases}$$

The repeated regression over a sequence of actions $\vec{a}$, denoted as $\mathcal{R}^*(\psi, \vec{a})$, is simply the repeated application of the regression operator through each action in the sequence (assuming it is defined at each step). For example, if $\vec{a} = [a_1, a_2, a_3]$ then the repeated regression is defined as $\mathcal{R}^*(\psi, \vec{a}) = \mathcal{R}(\mathcal{R}(\mathcal{R}(\psi, a_3), a_2), a_1)$.

Recall that in STRIPS, the goal formula is a conjunction of fluents which we typically represent as a set. Here, the set of fluents representing the goal is a partial state. Because of how we define regression in STRIPS, the result of regression, when defined, will also be a partial state. Therefore, the space and time complexity of regression in STRIPS remains linear in the number of fluents in the domain. The power of repeated regression is exemplified by the following theorem.

Theorem 2 (Regression Theorem (adapted from Reiter [86]))**.** *An action sequence $\vec{a}$ is a sequential plan for a planning problem $\langle F, O, I, G\rangle$ if and only if the following holds:*

$$I \models \mathcal{R}^*(G, \vec{a})$$

Theorem 2 provides a powerful method for determining precisely when a sequence of actions can achieve the goal. Further, if we are using the STRIPS formalism, the repeated regression of a goal formula represented as a conjunction of fluents is guaranteed to remain a conjunction of fluents. This property allows us to simplify the exposition of our techniques, as the repeated regression of a goal can be viewed as a partial state.

There is a key difference between the definition of STRIPS regression that we use and the one typically employed in automated planning. For $\mathcal{R}(\psi, a)$, we do not require that $ADD(a) \subseteq \psi$. This condition is typically used to ensure that only relevant actions are considered during "backward search" – a search technique that attempts to find a path from the goal to the initial state (as opposed to the opposite direction). For our purposes, we do not require this restriction on the applicability of regression.

2.3.2 Relevance in Execution Monitoring

There is a large body of research in execution monitoring (EM) in the planning community (e.g., De Giacomo *et al.* [27], Bjäreland [11], Pettersson [82], Fritz and McIlraith [38], and Doherty *et al.* [30]). We are concerned with EM systems that typically monitor the execution of a plan with the objective of ensuring that the plan is executing as intended. When something goes awry, the EM system takes an ameliorative action such as repairing the plan or replanning from scratch. In Chapters 4 and 5 we address the problem

of monitoring the execution of a POP, possibly including a set of temporal constraints imposed on the execution, with a view to exploiting the POP's inherent flexibility and robustness.

Exploiting the notion of regression, Fritz and McIlraith identified the conditions that ensure the validity of a sequential plan [38]: given a sequential plan $\vec{a} = [a_1, \ldots, a_n]$, the plan $\vec{a}$ remains valid with respect to the state of the world s, iff s entails one of the following conditions: $\mathcal{R}^*(G, [a_n])$, $\mathcal{R}^*(G, [a_{n-1}, a_n])$, $\ldots$, $\mathcal{R}^*(G, \vec{a})$. These conditions were integrated into the EM algorithm that checked the condition associated with each plan suffix, from the shortest to the longest suffix (i.e., the original plan) and resumed execution of the first suffix whose associated condition was entailed by the state (Definition 3, [38]). If no such condition was found, the EM system would decide to replan. We adopt a similar approach in Chapter 4 to monitor the continued viability of a POP.

In Chapter 5, we extend the notion of EM for a POP to include the continued satisfaction of a set of temporal constraints. The techniques used for temporal viability are based on the execution strategies of Simple Temporal Networks [28].

2.4 Summary

In this chapter, we have reviewed the prerequisite background for the remainder of the dissertation. We have also described work related to the overall approach we employ: focusing on what is relevant to improve the robustness and flexibility of plan representation, generation, and execution. Work related to each of the individual contributions is detailed in their respective chapters.

In the following chapter, we present the first core contribution of our work which focuses on deordering a sequential plan to produce a partial-order plan.

Chapter 3

Optimally Relaxing Partial-Order Plans

3.1 Introduction

For an agent to operate effectively in a dynamic world, its behaviour must be flexible in the face of unexpected changes. In the context of AI planning, there are a variety of approaches to improving the flexibility of an agent, including giving it the option to select from different plans [43] or improving the applicability of existing plans through plan generalization [3]. One example of the former is to delay committing to the ordering of certain actions in a plan until absolutely necessary, allowing the agent to dynamically choose how the plan proceeds at execution time [96]. This flexibility is precisely what *partial-order plans* provide.

Partial-order planning reflects a least commitment strategy [99]. Unlike a sequential plan, which specifies a set of actions and a total order over those actions, an ideal partial-order plan (POP) only specifies those action orderings necessary to achieve the goal. In doing so, a POP embodies a family of sequential plans – a set of linearizations all sharing the same actions, but differing with respect to the order of the actions. During execution, the agent is free to choose the next action to execute from the plan as long as the chosen action has no preceding actions left to execute. Increasing the number of linearizations in a plan translates directly to giving the agent more freedom at execution time.

The flexibility afforded by POPs makes them attractive for real-time execution, multi-agent taskability, and a range of other applications [96, 99]. Nevertheless, in recent years research on plan generation has shifted away from partial-order planning towards sequential planning, primarily due to the effectiveness of heuristic-based forward-search

planners. To regain the least commitment nature of POPs, while leveraging fast sequential plan generation, it is compelling to examine the computation of POPs via sequential planning technology (e.g., the forward-chaining partial-order planner POPF [18]).

In this chapter, we present an alternative approach that first generates a sequential plan with a state-of-the-art planner, and subsequently relaxes the plan to an optimally flexible POP. Deordering is the process of removing ordering constraints from a plan and reordering is the process of allowing any arbitrary change to the ordering constraints; both requiring that the POP remains valid. POP deordering and reordering have been theoretically investigated [4], and unfortunately optimal deordering or reordering is NP-hard to compute and difficult to approximate within a constant factor (unless $NP \in DTIME(n^{poly\ log\ n})$ [4]). Despite this theoretical impediment, we find that in practice we can often compute an optimal solution.

We focus on the problem of computing a *minimum deordering* and *minimum reordering* of a sequential plan. These notions cover a natural aspect of least commitment planning – minimizing the ordering constraints placed on a plan. Doing so naturally provides greater flexibility at execution time as there is an inverse correlation between the number of ordering constraints and the number of linearizations in a POP. We extend this characterization to additionally consider the total cost of actions in a plan – a metric commonly sought after as a measure of plan quality. We refer to a POP that has a minimum action cost (over the actions in the POP), and subsequently a minimum number of ordering constraints, as a *minimum cost least commitment POP* (MCLCP).[1] An MCLCP is compelling because it is free of any redundant actions and ordering constraints: *an MCLCP contains only what is relevant to achieve the goal.*

Our approach for computing an optimally relaxed POP is to use a family of novel encodings for partial weighted MaxSAT: an optimal solution to the MaxSAT problem corresponds to an optimally relaxed POP. Unlike typical SAT-based planning techniques, we represent an action occurrence once, giving us a succinct representation. We empirically compare our approach to an existing polynomial-time heuristic for relaxing a sequential plan due to Kambhampati and Kedar [53] and find that the latter is extremely proficient at computing a minimum deordering, matching the optimal solution in every problem tested. We find, however, that a minimum reordering is usually more flexible than a minimum deordering, having fewer ordering constraints and far more linearizations. We further see a reduction in the number of actions required for an MCLCP. We also compare the efficiency of our technique with a related approach that uses a Mixed Integer Linear

[1] Note that minimizing the total action cost in a uniform cost domain is equivalent to minimizing the number of actions.

Programming encoding to compute the minimum reordering. Our approach represents the first practical technique for computing the optimal reordering of a POP and optimal MCLCP, while proving the computed solution is optimal.

3.1.1 Contributions

The following are the main contributions of this chapter:

- We introduce a practical method for computing the optimal deordering and reordering of a plan. We accomplish this through a set of novel partial-weighted MaxSAT encodings, differing by a set of clause schema to define the type of relaxation we desire. We model the encodings after standard partial-order planning concepts, causal support and threat resolution, which we then draw upon to prove the correctness of our encodings.

- We propose an extension to least commitment planning, MCLCP, that includes the total cost of a solution. The optimization focuses first on minimizing the total action cost before maximizing the flexibility in the ordering constraints included in the plan. We further prove the correctness of our approach to use a partial weighted MaxSAT encoding for computing an MCLCP.

- We demonstrate, somewhat surprisingly, that an existing heuristic is extremely proficient at computing optimal deorderings. The existing algorithm produces only deorderings, and it is not theoretically guaranteed to find a minimal one, let alone an optimal deordering. Nonetheless, we find empirically that the heuristic computes the optimal deordering in every problem in our suite of benchmarks.

- We demonstrate the efficiency of our approach compared to a previous method that uses a similar encoding for a different optimization framework. For problems that are relatively difficult to relax (i.e., more than two seconds to compute), our approach improves on the previous work by solving 16% more of the problems within the given time bound.

- We show that we can achieve greater flexibility, compared to the optimal deordering, when using the optimal reordering or the introduced MCLCP criterion. This justifies the need for an approach such as ours to compute a more flexible plan.

The work in this chapter is based on the publications [75] and [77]. One notable change is the term for our introduced criterion, MCLCP. The publications simply use

the term LCP, which does not adequately reflect the MCLCP criterion. The evaluation was additionally expanded: experiments were given more resources and the comparison with a Mixed Integer Linear Programming approach was included.

3.1.2 Organization

We start by providing the necessary background and notation for partial weighted MaxSAT and forms of plan relaxation in Section 3.2. Next, we introduce the notion of ordering relevance in Section 3.3 and our new MCLCP criterion in Section 3.4. In Section 3.5 we present our family of encodings for computing deorderings and reorderings. Finally, we present our evaluation in Section 3.6 and conclude with a discussion of related work and summary in Section 3.7.

3.2 Preliminaries

3.2.1 Partial Weighted MaxSAT

In Boolean logic, the problem of Satisfiability (SAT) is to find a true/false setting of Boolean variables such that a logical formula referring to those variables evaluates to true [10]. Typically, we write problems in Conjunctive Normal Form (CNF), which is made up of a conjunction of clauses, where each clause is a disjunction of literals. A literal is either a Boolean variable or its negation. A setting of the variables satisfies a CNF formula iff every clause has at least one literal that evaluates to true. For example, setting variables x and z to be true will satisfy the following theory:

$$(\boldsymbol{x} \vee y) \wedge (\neg x \vee \boldsymbol{z}) \tag{3.1}$$

The MaxSAT problem is the optimization variant of the SAT problem in which the goal is to maximize the number of satisfied clauses [10, Ch. 19]. Although we cannot satisfy every clause in the following theory, setting x, y to true and z to false satisfies five clauses in the following theory:

$$(\boldsymbol{x} \vee \boldsymbol{y} \vee \boldsymbol{\neg z}) \wedge (\boldsymbol{x} \vee z) \wedge (\boldsymbol{y} \vee z) \wedge (\neg x \vee \neg y) \wedge (\boldsymbol{\neg z} \vee \neg x) \wedge (\boldsymbol{\neg z} \vee \neg y) \tag{3.2}$$

Adding non-uniform weights to each clause allows for a richer version of the optimiza-

tion problem, and we refer to optimizing with respect to the weight of satisfied clauses as the weighted MaxSAT problem. We use the syntax $(\overset{k}{\cdots})$ to indicate the clause has a weight of k. Generally, the weight must be a positive real number. Consider setting x to false and y, z to true in the following theory:

$$(\overset{3}{x}) \wedge (\boldsymbol{\neg x} \overset{1}{\vee} \neg y) \wedge (\boldsymbol{\neg x} \overset{1}{\vee} \neg z) \wedge (\overset{1}{\boldsymbol{y}}) \wedge (\overset{1}{\boldsymbol{z}}) \tag{3.3}$$

While the setting satisfies four clauses, it only has a total weight of 4. With the aim of maximizing the total weight of satisfied clauses, we can achieve a sum of 5 by assigning all variables to true:

$$(\overset{3}{\boldsymbol{x}}) \wedge (\neg x \overset{1}{\vee} \neg y) \wedge (\neg x \overset{1}{\vee} \neg z) \wedge (\overset{1}{\boldsymbol{y}}) \wedge (\overset{1}{\boldsymbol{z}}) \tag{3.4}$$

If we wish to force the solver to find a solution that satisfies a particular subset of the clauses, we refer to clauses in this subset as *hard*, while all other clauses in the problem are *soft*. The syntax we use to indicate a hard clause is $(\overset{\bullet}{\cdots})$. When we have a mix of hard and soft clauses, we have a partial weighted MaxSAT problem [10, Ch. 19.6].

In a partial weighted MaxSAT problem, only the soft clauses are given a weight, and a feasible solution corresponds to any setting of the variables that satisfies the hard clauses in the CNF. An optimal solution to a partial weighted MaxSAT problem is any feasible solution that maximizes the sum of the weights on the satisfied soft clauses. In the following example, setting variables x, y to false and z to true satisfies every hard clause and one of the soft clauses:

$$(\overset{1}{x}) \wedge (\overset{2}{y}) \wedge (\overset{3}{\boldsymbol{z}}) \wedge (\boldsymbol{\neg x} \overset{\bullet}{\vee} \neg z) \wedge (\boldsymbol{\neg y} \overset{\bullet}{\vee} \neg z) \wedge (\boldsymbol{\neg x} \overset{\bullet}{\vee} \boldsymbol{\neg y}) \tag{3.5}$$

Although not required for partial weighted MaxSAT in general, the encodings we create will never contain a soft clause that has more than one literal. This special form of partial weighted MaxSAT problem, referred to as a binate covering problem [21], allows us to flip the optimization criteria: minimizing the sum of the satisfied soft (unit) clauses is equivalent to maximizing the sum of unit clauses that have the literal flipped (e.g., x goes to $\neg x$ and vice versa). Using this technique to solve the minimization problem with a partial weighted MaxSAT solver only works if the soft clauses contain a single literal. This property is key to our encoding, as our objective is always to minimize.

3.2.2 Deorderings and Reorderings

The aim of least commitment planning is to find flexible plans that allow us to defer decisions regarding the execution of the plan. Considering only the ordering constraints of a POP, two important notions for least commitment planning are the *deordering* and *reordering* of a POP. Following Bäckström [4], we define these formally:

Definition 3 (Deordering and Reordering)**.** Let $P = \langle \mathcal{A}, \mathcal{O} \rangle$ and $Q = \langle \mathcal{A}', \mathcal{O}' \rangle$ be two POPs, and Π a STRIPS planning problem:

1. Q is a deordering of P wrt. Π iff P and Q are valid POPs for Π, $\mathcal{A} = \mathcal{A}'$, and $\mathcal{O}' \subseteq \mathcal{O}$.

2. Q is a reordering of P wrt. Π iff P and Q are valid POPs for Π, and $\mathcal{A} = \mathcal{A}'$.

Recall that we assume the ordering constraints of a POP to be transitively closed, and every action in a POP is uniquely named (i.e., every repetition of the same action is given a unique name). Although we do not consider it further here, a *proper* deordering is one where the ordering constraints are different (i.e., $\mathcal{O}' \subsetneq \mathcal{O}$). We define the minimum deordering and minimum reordering as follows:

Definition 4 (Minimum Deorderings and Reorderings)**.** Let $P = \langle \mathcal{A}, \mathcal{O} \rangle$ and $Q = \langle \mathcal{A}', \mathcal{O}' \rangle$ be two POPs, and Π a STRIPS planning problem:

1. Q is a minimum deordering of P wrt. Π iff

   (a) Q is a deordering of P wrt. Π, and

   (b) there is no deordering $\langle \mathcal{A}'', \mathcal{O}'' \rangle$ of P wrt. Π s.t. $|\mathcal{O}''| < |\mathcal{O}'|$.

2. Q is a minimum reordering of P wrt. Π iff

   (a) Q is a reordering of P wrt. Π, and

   (b) there is no reordering $\langle \mathcal{A}'', \mathcal{O}'' \rangle$ of P wrt. Π s.t. $|\mathcal{O}''| < |\mathcal{O}'|$.

Note that we use $<$ rather than $\subset$ for 1(b) and 2(b) because the orderings in $\mathcal{O}'$ and $\mathcal{O}''$ need not overlap. We will equivalently refer to a minimum deordering (resp. reordering) as an *optimal* deordering (resp. reordering). In both cases, we prefer a POP that has the smallest set of ordering constraints. In other words, no POP exists with the same actions and fewer ordering constraints while remaining valid with respect to Π. The problem of finding a minimum deordering or reordering of a POP is NP-hard, and cannot be approximated within a constant factor unless $NP \in \text{DTIME}(n^{poly\ log\ n})$ [4].

The final notion of interest is a *minimal* deordering: Q is a minimal deordering of P iff Q is a deordering of P and there does not exist a proper deordering of Q. We can compute a minimal deordering in polynomial time by iteratively removing unnecessary ordering constraints (i.e., those that do not cause the plan to become invalid).

3.2.3 Previous Approaches

There are a variety of approaches to computing a partial-order plan, and we cover the spectrum of them here.

Partial-Order Causal Link Algorithms

Traditional methods for producing a partial-order plan follow an approach called *partial-order causal link* (POCL) planning [99]. In POCL planning, modifications are iteratively made to an incomplete partial-order plan that consists of a set of actions, causal links, and ordering constraints. A partial-order plan is considered complete if and only if three conditions are met (cf. Section 2.2.2 for terminology)

1. The plan contains the special actions a_I and a_G.
2. There is no open precondition for any action in the plan.
3. Every causal link is threat free.

The key difference between POCL planning and the standard state-based search, is that POCL planning is a search through plan space – every step in the search space constitutes a partially instantiated plan. Possible modifications to the partial plan represent the choices available in the POCL planning search procedure. The typical modifications permitted in POCL planning include the following:

1. Add a new action to the partial plan.
2. Order two actions in the partial plan.
3. Create a causal link between two actions in the plan.

POCL planners were popular in the 1980s and 1990s until forward search techniques, such as the one employed by the FF planner [51], led the planning research in a new direction. The most recent POCL planner is VHPOP [104], but unfortunately it is not competitive with the state-of-the-art forward search planners.

POPF

To take advantage of the flexibility afforded by a POP and the search efficiency of forward state-based planners, Coles *et al.* introduced the forward-chaining partial-order planner POPF [18]. The idea behind POPF is to restrict the modifications permitted to the partially completed plan so that a complete state can be easily computed that represents the truth of fluents after the partial plan is executed. Unlike POCL approaches, the only open preconditions permitted are those for the goal action. When a new action is added to the plan, it is placed at the end of the plan – no action already in the plan (aside from the special goal action) can be ordered *after* the newly added action. Further, adding a new action requires that all of its preconditions have causal links created immediately.

The approach used in POPF leverages the partial-order nature of planning domains by avoiding unnecessary reasoning about the permutation of certain unordered actions. Sequential planners may try to complete the same partial-order plan multiple times with the only change being a different permutation of unordered actions. Additionally, POPF leverages the powerful techniques of forward-search planners by maintaining the complete state of the world that will be reached by the plan. Having this state information allows for powerful heuristics to be computed efficiently.

Relaxer Algorithm

Due to Kambhampati and Kedar [53], the Relaxer Algorithm[2] operates by removing ordering constraints from a sequential plan in a systematic manner. A heuristic guides the procedure, and as pointed out by Bäckström [4], the process does not provide any guarantee the resulting POP is minimally deordered. There is an error in the counterexample used by Bäckström to demonstrate a minimally deordered POP is not produced by Kambhampati and Kedar's algorithm. However, the conclusion is correct, and we provide a new counterexample in Appendix A.

The intuition behind the algorithm is to remove any ordering $(a_i \prec a_k)$ from the sequential plan where a_i is not the achiever of some precondition of a_k and removing the ordering does not lead to a threat. The algorithm heuristically attempts to choose the earliest possible action in the sequential plan as the achiever of a precondition. For example, consider the case where our sequential plan is $[a_1 \cdots, a_i, \cdots, a_k, \cdots, a_n]$ and $p \in PRE(a_k)$. The algorithm will keep the ordering $(a_i \prec a_k)$ only if leaving it out would create a threat for a precondition of one of the actions, or if a_i is the earliest action in the sequence where the following holds:

[2]Referred to as "order generalization" originally.

1. $p \in ADD(a_i)$: a_i is an achiever for p

2. $\forall a_j, i < j < k, p \notin DEL(a_j)$: p is not threatened.

Algorithm 4, which we will refer to as the *Relaxer Algorithm*, presents this approach formally. We use **index**(a,$\vec{a}$) to refer to the index of action a in the sequence $\vec{a}$, and assume every action in the plan is uniquely named.

Algorithm 4: Relaxer Algorithm

Input: Sequential plan, $\vec{a}$, including a_I and a_G
Output: Relaxed Partial-order plan, $\langle \mathcal{A}, \mathcal{O} \rangle$

```
1  A = set(a⃗);
2  O = ∅;
3  foreach a ∈ A do
4      foreach f ∈ PRE(a) do
5          ach = null;
6          for i = (index(a, a⃗) − 1) ··· 0 do
7              // See if we have an earlier achiever
8              if f ∈ ADD(a⃗[i]) then
9                  ach = a⃗[i];
10             // Stop if we find a deleter of f
11             if f ∈ DEL(a⃗[i]) then
12                 break;
13         // Add the appropriate supporting link
14         O = O ∪ {(ach ≺ a)};
15         // Add orderings to avoid threats
16         foreach a′ ∈ deleters(f) \ {a} do
17             if index(a′, a⃗) < index(ach, a⃗) then
18                 O = O ∪ {(a′ ≺ ach)};
19             if index(a′, a⃗) > index(a, a⃗) then
20                 O = O ∪ {(a ≺ a′)};
21 return ⟨A, O⟩;
```

If $\vec{a}$ is a valid plan, line 8 will evaluate to true before either line 11 evaluates to true or the for-loop at line 6 runs out of actions. That is, we know an unthreatened achiever exists and the earliest such one is found. Note, however, that the achiever may also delete the fluent. We allow this, as the standard interpretation is the action first deletes the fluent and then adds it, which does not prohibit the action from being an achiever.

The achiever is then ordered before the action requiring the fluent as a precondition (line 14), and the for-loop at line 16 adds all of the necessary ordering constraints so the

achiever remains unthreatened. Note that for any deleter found in this for-loop, either line 17 or 19 must evaluate to true.

After going through the outer loop at line 3, every action in the newly formed POP has an unthreatened supporting action for each of its preconditions. The resulting POP will therefore be valid (cf. Section 5.2 of Kambhampati and Kedar [53]).

SAPA Post-Processing

As part of a post-processing phase for the SAPA planner, Do and Kambhampati introduce an approach similar to ours for relaxing the ordering of a plan [29]. In their setting, they begin with a temporal plan with the actions assigned to specific time points, and the objective is to optimize either the number of ordering constraints or some temporal aspect of the resulting plan.

The strategy Do and Kambhampati take (abbreviated as DK here), is to model the task of computing a partial-order relaxation in terms of a constraint satisfaction optimization problem (CSOP). Variables are introduced to represent the ordering of actions, the timing and duration of actions, the resource usage, etc. From the abstract CSOP formalism, a concrete mixed integer linear program (MILP) is provided to realize the set of constraints that model a valid temporal plan. Similar to our work, DK contains the option for enforcing adherence to the original ordering constraints which allows either a deordering or a reordering to be produced.

DK considers a number of optimization criteria including minimizing the makespan, maximizing the sum of slack in the temporal variables, maximizing the flexibility in the temporal variables, and minimizing the number of ordering constraints. While the first three are related to temporal planning domains, the final one coincides with the optimization criteria of our work. Experimental evaluation is provided for the temporal optimization criterion, but Do and Kambhampati do not empirically investigate the minimization of ordering constraints.

Differences between DK and our approach include the formalism (we do not focus on temporal aspects), the model used (unique to our encoding are variables that represent an action appearing in the plan and unique to their encoding are variables representing time points and resources), the underlying solving technology (we rely on partial weighted MaxSAT instead of MILP), and most importantly the MCLCP criteria. Within a single optimization criteria, MCLCP aims to minimize both the total action cost as well as the number of ordering constraints. In Section 3.6.4 we compare the efficiency of our approach for computing a minimum reordering with an implementation of the DK approach that uses only the variables and constraints relevant to computing a minimum reordering.

3.3 Ordering Relevance

We can view a sequential plan as a special case of a partial-order plan where there exists an ordering constraint between every pair of actions. Quite often, many of these ordering constraints are not required: the ordering of certain actions may be switched and the goal still achieved with the new sequence of actions. With the aim of maximizing the flexibility of a POP, we strive to minimize the number of ordering constraints included in the solution. This objective motivates the need to identify precisely which ordering constraints in a POP are relevant to the POP's validity.

Definition 5 (Ordering Relevance)**.** Given a planning problem $\Pi = \langle F, O, I, G\rangle$ and valid POP $P = \langle \mathcal{A}, \mathcal{O}\rangle$ for Π, the ordering constraint $o \in \mathcal{O}$ is *relevant* with respect to Π and P iff $\langle \mathcal{A}, \mathcal{O} - \{o\}\rangle$ is not a valid POP for Π.[3]

Ordering relevance plays a central role in the definitions of minimal and minimum POP deorderings: the relevant ordering constraints are precisely those that cannot be removed without invalidating the POP [4]. Additionally, the Relaxer Algorithm of Kambhampati and Kedar [53] operates by identifying a set of ordering constraints suspected of being relevant.

3.4 Minimum Cost Least Commitment Criteria

While the notion of a minimum deordering or reordering of a POP addresses the commitment of ordering constraints, an orthogonal objective is to commit as few resources as possible – typically measured as the sum of action costs for the actions in a plan. Historically, this objective takes precedence over all other metrics. To this end, we provide the extended criterion minimum cost least commitment POP (MCLCP).

Definition 6 (Minimum Cost Least Commitment POP)**.** Let $P = \langle \mathcal{A}, \mathcal{O}\rangle$ and $Q = \langle \mathcal{A}', \mathcal{O}'\rangle$ be two POPs valid for Π. Q is a *minimum cost least commitment POP* (MCLCP) of P iff Q is a minimum reordering of itself, $\mathcal{A}' \subseteq \mathcal{A}$, and there does not exist a valid POP $R = \langle \mathcal{A}'', \mathcal{O}''\rangle$ for Π such that the following conditions hold:

$$\mathcal{A}'' \subseteq \mathcal{A}$$
$$c_\Pi(R) < c_\Pi(Q)$$

[3] Note that the transitive closure of P is necessarily different from the transitive closure of $\langle \mathcal{A}, \mathcal{O} - \{o\}\rangle$ when o is relevant with respect to Π and P.

For this work, we assume that every action in Π has positive cost. Intuitively, we can compute an MCLCP of an arbitrary POP by first minimizing the total action cost, and then minimizing the number of ordering constraints. It may turn out that preferring fewer actions causes us to commit to more ordering constraints, simply due to the interaction between the actions we choose. In practice, however, we usually place a much greater emphasis on minimizing the total cost of a plan. It is also worth noting that if no plan exists with a subset of the actions in the input plan, computing the MCLCP is equivalent to computing a minimum reordering.

Following the above criteria, we evaluate the quality of a POP by the total action cost and number of ordering constraints it contains; these metrics give us a direct measure of the least commitment nature of a POP. While difficult to measure efficiently, we strive to maximize the *flexibility* inherent in a POP; defined as the number of linearizations a POP represents. The number of unordered pairs of actions in a POP, typically referred to as *flex*, provides an approximation for the POP's flexibility [92].

As we have discussed earlier, verifying a POP's validity by way of the linearizations is not always practical. Similarly, we will not attempt to compute POPs that maximize the number of linearizations, but rather we will compute POPs that adhere to one of the previously mentioned criteria: minimum deordering, minimum reordering, or MCLCP.

3.5 Encoding

We encode the task of finding a minimum deordering or reordering as a partial weighted MaxSAT problem. An optimal solution to the default encoding will correspond to a MCLCP. That is, no POP exists with a cheaper overall cost, or with fewer ordering constraints. We present this core encoding in Section 3.5.1 and prove the soundness and completeness of the encoding in Section 3.5.2. We add further clauses to produce encodings that correspond to optimal deorderings or reorderings, and present these modifications in Section 3.5.3.

3.5.1 Basic Encoding

In contrast to the typical SAT encoding for a planning problem (e.g., Kautz and Selman [55]), we do not require that the actions be replicated for successive plan steps. Instead, we represent each action occurrence only once and reason about the ordering between actions. The actions in the encoding come from a provided sequential or partial-order plan, $P = \langle \mathcal{A}, \mathcal{O} \rangle$. We use $\Phi(P)$ to denote the partial weighted MaxSAT encoding

corresponding to the POP $P = \langle \mathcal{A}, \mathcal{O} \rangle$, and refer to the POP corresponding to an encoding's solution as the *target POP*. A target POP can be reconstructed from an encoding's solution by looking at only the variables set to true. We use three types of propositional variables:

- x_a: For every action a in $\mathcal{A}$, x_a indicates that action a appears in the target POP.
- $\kappa(a_1, a_2)$: For every pair of actions a_1, a_2 in $\mathcal{A}$, $\kappa(a_1, a_2)$ indicates that the ordering constraint $(a_1 \prec a_2)$ appears in the target POP.
- $\Upsilon(a_i, p, a_j)$: For every action a_j in $\mathcal{A}$, p in $\text{PRE}(a_j)$, and a_i in $\mathbf{adders}(p)$, $\Upsilon(a_i, p, a_j)$ indicates a_i supports a_j with the fluent p in the target POP.

In a partial weighted MaxSAT encoding there is a distinction between hard and soft clauses. We first present the hard clauses of the encoding as Boolean formulae which we subsequently convert to CNF, and later describe the soft clauses with their associated weight.[4] We define the formulae that ensure that the target POP is acyclic, and the ordering constraints include the transitive closure. Here, actions are universally quantified, and for formula (3.9) we assume $a_I \neq a_i \neq a_G$. We must ensure that:

- There are no self-loops:

$$(\neg\kappa(a, a)) \tag{3.6}$$

- We include the initial and goal actions:

$$(x_{a_I}) \wedge (x_{a_G}) \tag{3.7}$$

- If we use an ordering variable, then we include both actions:

$$\kappa(a_i, a_j) \rightarrow x_{a_i} \wedge x_{a_j} \tag{3.8}$$

- An action cannot appear before the initial action (or after the goal):

$$x_{a_i} \rightarrow \kappa(a_I, a_i) \wedge \kappa(a_i, a_G) \tag{3.9}$$

[4]For readability, we omit the hard clause symbol, $(\overset{\bullet}{\cdots})$, for constraints (3.6)-(3.12).

- A solution satisfies the transitive closure of ordering constraints:

$$\kappa(a_i, a_j) \wedge \kappa(a_j, a_k) \rightarrow \kappa(a_i, a_k) \tag{3.10}$$

Together, (3.6) and (3.10) ensure that the target POP will be acyclic (note that this implies antisymmetry as well), while the remaining formulae tie the two types of variables together and deal with the initial and goal actions. Finally, we include the formulae needed to ensure that every action has its preconditions met, and there are no threats in the solution:

$$\Upsilon(a_i, p, a_j) \rightarrow \bigwedge_{a_k \in \mathbf{deleters}(p)} x_{a_k} \rightarrow \kappa(a_k, a_i) \vee \kappa(a_j, a_k) \tag{3.11}$$

$$x_{a_j} \rightarrow \bigwedge_{p \in PRE(a_j)} \bigvee_{a_i \in \mathbf{adders}(p)} \kappa(a_i, a_j) \wedge \Upsilon(a_i, p, a_j) \tag{3.12}$$

Intuitively, $\Upsilon(a_i, p, a_j)$ ensures that if a_i is the achiever of precondition p for action a_j, then no deleter of p will be allowed to occur between the actions a_i and a_j. Formula (3.11) ensures that every causal link remains unthreatened in a satisfying variable setting, and we can view the two ordering variables in the formula as a form of the common partial-order planning concepts of promotion and demotion [99]. Formula (3.12) ensures that if we include action a_j in the POP, then every precondition p of a_j must be satisfied by at least one achiever a_i. $\kappa(a_i, a_j)$ orders the achiever correctly, while $\Upsilon(a_i, p, a_j)$ removes the possibility of a threatening action.

To generate an MCLCP, we prefer solutions that first minimize the total action cost, and then minimize the number of ordering constraints. We achieve this by adding a soft unit clause for every action and ordering variable in our encoding which contains the negation of the variable. A violation of any one of the unit clauses means that the solution includes the action or ordering constraint corresponding to the violated clause's variable. The weight assigned is as follows:

- $(\neg\overset{1}{\kappa}(a_i, a_j)), \ \forall a_i, a_j \in \mathcal{A}$
- $(\neg x_a)^{c_\Pi(a)+|\mathcal{A}|^2+1}, \ \forall a \in \mathcal{A} \setminus \{a_I, a_G\}$

Note that the weight of any single action clause is greater than the weight of all ordering constraint clauses combined. The increased weight guarantees that we generate

solutions with a minimum action cost.[5] Because we enforce the transitive closure of the ordering constraints, the second type of soft clause will lead the solver to find a POP (among those with the cheapest total action cost) that minimizes the size of the transitive closure.

3.5.2 Theoretical Results

In this section we present theoretical properties of our core encoding.

Lemma 1 (Variable Setting Implies POP)**.** *Given a planning problem* Π *and a valid POP* $P = \langle \mathcal{A}, \mathcal{O} \rangle$*, any variable setting that satisfies the formulae (3.6)-(3.12) for* $\Phi(P)$ *will correspond to a valid POP for* Π *where the ordering constraints are transitively closed.*

Proof. We have already seen that the POP induced by a solution to the hard clauses will be acyclic and transitively closed (due to formulae (3.6)-(3.10)). We can further see that there will be no open preconditions because we include a_G, and the conjunction of (3.12) ensures that every precondition will be satisfied when the POP includes an action. Additionally, there are no threats in the final solution because of formula (3.11), which will be enforced every time a precondition is met by formula (3.12). Because the POP corresponding to any solution to the hard clauses will have no open preconditions and no threats, Theorem 1 allows us to conclude that the target POP will be valid for Π. □

Lemma 2 (POP Implies Variable Setting)**.** *Given a planning problem* Π *and a valid POP* $P = \langle \mathcal{A}, \mathcal{O} \rangle$*, any valid POP* $Q = \langle \mathcal{A}', \mathcal{O}' \rangle$*, where* $\mathcal{A}' \subseteq \mathcal{A}$ *and* $\mathcal{O}'$ *is transitively closed, has a corresponding variable assignment that satisfies* $\Phi(P)$*.*

Proof. The lemma follows from the direct encoding of the POP Q where $x_a = true$ iff $a \in \mathcal{A}'$ and $\kappa(a_i, a_j) = true$ iff $(a_i \prec a_j) \in \mathcal{O}'$. If Q is a valid POP, then it will be acyclic, include a_I and a_G, have all actions ordered after a_I and before a_G, and be transitively closed (satisfying (3.6)-(3.10)). We further can see that (3.11) and (3.12) must be satisfied: if (3.12) did not hold, then there would be an action a in the POP with a precondition p such that *every* potential achiever of p has a threat that could be ordered between the achiever and a. Such a situation is only possible when the POP is invalid, which is a contradiction. □

Theorem 3 (Completeness)**.** *Given a planning problem* Π *and a valid POP* $P = \langle \mathcal{A}, \mathcal{O} \rangle$*, a complete partial weighted MaxSAT solver will find a solution to the soft clauses and formulae (3.6)-(3.12) for* $\Phi(P)$ *that minimizes the total cost of actions in the corresponding POP, and subsequently minimizes the number of ordering constraints.*

[5] If we wish to minimize the number of actions in the solution, we need only to replace $c_\Pi(a)$ with 0.

Proof. Given $|\mathcal{A}|$ actions, there can only be $|\mathcal{A}|^2$ ordering constraints. Because every soft clause corresponding to an ordering constraint has a weight of 1, the total sum of satisfying every ordering constraint clause will be $|\mathcal{A}|^2$. Because the weight of satisfying any action clause is greater than $|\mathcal{A}|^2$, the soft clauses corresponding to actions dominate the optimization criteria. As such, there will be no valid POP for Π which has a subset of the actions in P with a lower total action cost than a solution that satisfies formulae (3.6)-(3.12) while maximizing the weight of the satisfied soft clauses. □

Theorem 4 (Encoding Correctness)**.** *Given a planning problem Π, and a valid POP P for Π, a solution to our partial weighted MaxSAT encoding $\Phi(P)$ is an MCLCP for P.*

Proof. This Theorem follows directly from Lemmas 1, 2, and Theorem 3. □

3.5.3 Variations

Observe that $\Phi(P)$ does not make use of the set of ordering constraints in P. An optimal solution to the encoding will correspond to an MCLCP, but to enforce solutions that are minimum deorderings or reorderings, we introduce two additional sets of hard clauses.

All Actions

For optimal deorderings and reorderings, we require every action to be a part of the target POP. We consider a formula that ensures that we use every action (and so the optimization works only on the ordering constraints). To achieve this, we simply need to add each action as a hard clause:

$$\overset{\bullet}{(x_a)}, \ \ \forall a \in \mathcal{A} \tag{3.13}$$

An optimal solution to the soft constraints and formulae (3.6)-(3.13), referred to as $\Phi^{MR}(P)$, corresponds to a minimal reordering of P.

Deordering

For a deordering we must forbid any explicit ordering that contradicts the input plan. Assuming our input plan is $P = \langle \mathcal{A}, \mathcal{O} \rangle$, we ensure that the computed solution is a deordering by adding the following family of hard unit clauses:

$$\overset{\bullet}{(\neg\kappa(a_i, a_j))}, \ \ \forall (a_i \prec a_j) \notin \mathcal{O} \tag{3.14}$$

An optimal solution to the soft constraints and formulae (3.6)-(3.14), referred to as $\Phi^{MD}(P)$, corresponds to a minimal deordering of P. We additionally could use (3.14) and forgo the use of (3.13), but this variation is not one typically studied, nor does it provide a benefit over computing an MCLCP.

3.6 Evaluation

We evaluate the effectiveness of using the state-of-the-art partial weighted MaxSAT solver, Sat4j [67], to optimally relax a plan using our proposed encodings. We additionally evaluated the 2013 winner of the partial weighted MaxSAT contest for crafted instances, MaxHS [25]. However, we found that Sat4j slightly outperformed MaxHS in both coverage and time. To measure the quality of the POPs we generate, we consider the total cost of actions in the plan, the number of ordering constraints, and the number of linearizations (whenever feasible to compute). Furthermore, we investigate the effectiveness of the Relaxer Algorithm to produce a minimally constrained deordering.

For our analysis, we use six representative domains from the International Planning Competition (IPC)[6] that allow for a partially ordered solution: depots, driverlog (driver), logistics, tpp, rovers, and zenotravel (zeno). These domains allow for partial-order solutions – many IPC domains are constrained so as to only yield sequential plans (e.g., Sokoban). In such cases, relaxation is not possible and the solver trivially finds the sequential plan we begin with. We conducted the experiments on a Linux desktop with a 2.13GHz processor, and each run of Sat4j was limited to 30 minutes and 2GB of memory.

We generated an initial sequential plan by using the FF planner [51]. The encodings provided in Section 3.5 were converted to CNF using simple distributive rules so they may be used with Sat4j. We investigated the possibility of using a starting solution from POPF [18], but found that POPF failed to solve as many of the problems as FF. Of the problems that were mutually solved by FF and POPF, we found that the POPs produced by the POPF planner were quite over-constrained, having many more ordering constraints than necessary. That is to say, FF+Sat4j produces far more relaxed plans than POPF – several orders of magnitude fewer ordering constraints in some cases. Once relaxed, the optimal deordering, optimal reordering, and MCLCP of both FF and POPF plans were very similar: there is usually little difference between the POPs generated by relaxing a sequential FF plan and the POPs generated by relaxing the POP found by POPF. For this reason, we only present the results for FF.

In the following evaluation, we only report on the problems where FF was able to

[6] `http://ipc.icaps-conference.org/`

find a plan within a 30-minute timeout.[7] We assess various aspects of our approach through three separate experiments. First, we evaluate the quality of the POP produced by our encodings as well as the POP produced by the Relaxer Algorithm. Here, quality is measured both as the total action cost and number of ordering constraints in the transitive closure. Next, we consider the difficulty of solving the encoding with Sat4j by observing the number of problems that would be solved if given a particular time limit. We then evaluate the potential improvement in flexibility when using a minimum reordering of a plan in lieu of a minimum deordering of a plan. Finally, we compare our approach for computing a minimum reordering with that of Do and Kambhampati [29].

3.6.1 POP Quality

We begin by examining the relative quality of the POPs produced with different optimization criteria corresponding to the various encodings we have proposed – MCLCP (Φ), MR (Φ^{MR}), and MD (Φ^{MD}) – as well as the Relaxer Algorithm (abbreviated as RX). Quality is measured by the total action cost and number of ordering constraints in the transitive closure of the generated POP. The total action cost for RX, MR, and MD are equal to those in the sequential plan, so we report the value only for RX and MCLCP. Table 3.1 shows the results for all six domains on the problems for which every approach succeeded in finding a solution (98 of the 130 successfully encoded problems).

There are a few items of interest to point out. First, columns 4 and 5 coincide perfectly. Perhaps surprisingly, RX is able to produce the optimal deordering in every case, even though it is not guaranteed to do so. Because the algorithm can only produce deorderings, this is the best we could hope for from the algorithm. Second, we see the number of ordering constraints for the MCLCP approach is greater, on average, than that for the MR approach in the logistics domain. The reason for this is that POPs in the logistics domain require more ordering constraints for a solution with slightly fewer actions. For example, in one of the problem instances the MR solution has 100 actions and 2278 ordering constraints, while the MCLCP solution reduces the number of actions in the POP to 96 at the expense of requiring 2437 ordering constraints. Third, the low coverage in the tpp domain is due to allowing reordering – for example, MD succeeds in 20 problems while MCLCP and MR succeed in only 5.

Generally, the MCLCP has fewer actions and ordering constraints than the optimal reordering, which in turn has fewer ordering constraints than the optimal deordering. If the MCLCP has the same number of actions as the sequential plan, then the MCLCP

[7] Providing twice the amount of time to the FF solver did not allow further problems to be solved.

Domain	$\sum_{a \in \mathcal{A}} c_{\Pi}(a)$		$\lvert\mathcal{O}\rvert$			
	RX	MCLCP	RX	MD	MR	MCLCP
depots (14/22)	34.9	31.0	473.4	473.4	430.9	341.5
driver (15/16)	27.5	26.5	332.6	332.6	326.9	297.3
logistics (30/35)	78.1	77.4	1490.6	1490.6	1462.5	1470.4
tpp (5/30)	13.4	13.4	74.8	74.8	74.8	74.8
rovers (18/20)	31.1	30.3	223.2	223.2	217.6	204.2
zeno (16/20)	29.2	29.2	404.3	404.3	403.5	403.5
ALL (98/143)	44.3	43.2	685.7	685.7	669.0	651.6

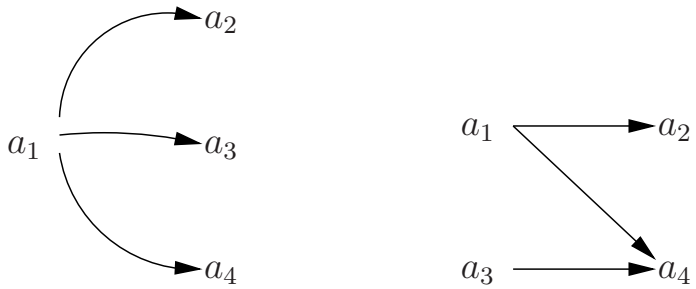
Table 3.1: Mean total action cost and number of ordering constraints for the various approaches. Numbers next to the domain indicate the number of instances solved by all methods (and included in the mean). Note that the mean action cost uses the original action cost in the domain and not the weight of the unit clauses corresponding to actions.

and minimum reordering coincide. We can see this effect in tpp and zenotravel, where the number of actions and ordering constraints for MCLCP and MR are equivalent.

Finally, we note that in 4 problems (1 from logistics, and 3 from rovers), we found that the number of actions and constraints, for either the MCLCP or MR POP, matched that of the Relaxer POP, but the number of linearizations differed. Further, the differences in the number of linearizations were not all in the same direction (some better, and some worse). While the number of ordering constraints in a POP (for a given number of actions) usually gives an indication of the number of linearizations for that POP, these 4 problems indicate that this is not always the case.

For a concrete example, consider two POPs on four actions $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$. Ignoring causal links, Figure 3.1 shows the structure of the POPs P_1 and P_2. Both POPs have the same number of actions and ordering constraints, but the number of linearizations differ: P_1 has 6 linearizations while P_2 only has 5. These POPs serve as a basic example of how the MCLCP criteria does not fully capture the notion of POP flexibility. We should point out, however, that fewer ordering constraints usually

indicates more linearizations.



(a) P_1 (b) P_2

Figure 3.1: Two POPs with the same number of actions and ordering constraints, but different number of linearizations.

3.6.2 Encoding Difficulty

To assess the difficulty of solving the encoded problems, we show the number of problems solved by Sat4j as a function of time (including the encoding phase) in Figure 3.2. Sat4j consistently produced a feasible solution almost immediately (under a second in the majority of problems). While we do not consider it here, early solutions serve as approximations to the optimal plan and are valid POPs (Sat4j finds the optimal solution if given sufficient time and memory). For comparison, we additionally include the aggregate time for RX. When Sat4j had difficulty, it was due to the number of transitivity clauses included for formula 3.10, which are cubic in the number of actions.

We find that MD is generally easier to solve and the majority of the problems are readily handled by Sat4j: 71% solved in under 10 seconds. Being a polynomial algorithm, RX consistently deorders the plan faster than any encoding.

3.6.3 Reordering Flexibility

We have already seen that the Relaxer Algorithm is capable of producing a minimum deordering. To assess the potential gain in flexibility for using a minimum reordering, we compare the number of linearizations the Relaxer Algorithm induces with the number of linearizations the optimal reordering induces. We found that of the 78 problems for which we could successfully compute the linearizations, approximately 40% of the problems exhibited a difference between the optimal deordering and optimal reordering. Figure 3.3 shows the number of linearizations for the optimal reordering divided by the

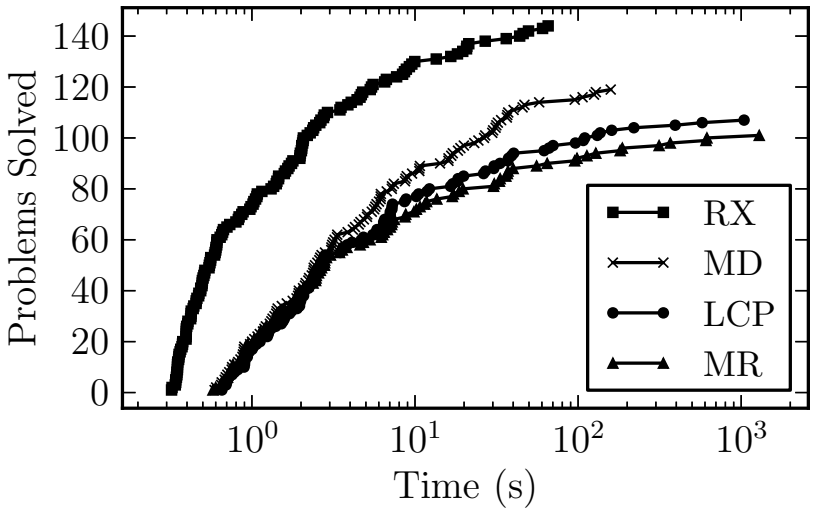


Figure 3.2: The number of problems solved by Sat4j if given a limited amount of time per problem.

number of linearizations for the optimal deordering. For readability, we omit the 47 instances where the linearizations matched.

The ratio of linearizations ranges from 0.9 to over two million. The single ratio under 1 occurs due to the rare case in which two valid POPs with an equal number of actions and ordering constraints have a different number of linearizations. While the Relaxer Algorithm is proficient at finding the optimal deordering, this result demonstrates that there is still a significant advantage in terms of flexibility when using the optimal reordering.

3.6.4 Comparison to MILP Encoding

The model for relaxing the ordering of a plan that is presented by Do and Kambhampati involves temporal constraints and resources [29] – both are aspects beyond what we consider here. Nevertheless, a fragment of the model is capable of computing either the minimum deordering or reordering of the plan, and so it is worthwhile to see how effective it can be in finding an optimal reordering. We forgo testing the previous work for computing the optimal deordering, as the Relaxer Algorithm is so effective in doing so. We should note that the original work only considered using the model to heuristically guide the solver to a reasonable solution instead of an optimal one.

The optimization framework Do and Kambhampati use to model the problem of relaxing the ordering of a plan is Mixed Integer Linear Programming (MILP) [89]. Classically, a MILP consists of a set of linear constraints that are defined over variables that

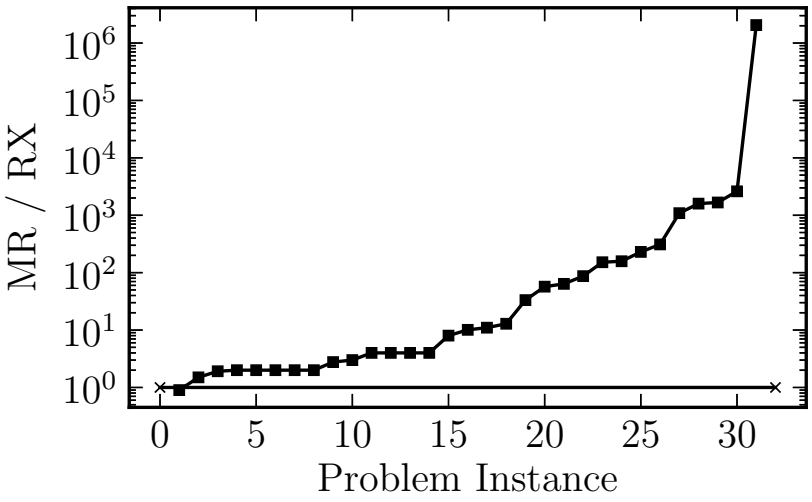


Figure 3.3: Ratio of Linearizations. The y-axis represents the number of linearizations induced by the POP for the optimal reordering divided by the number of linearizations induced by the POP for the optimal deordering. The x-axis ranges over all problems where the number of linearizations differed (∼40%), and is sorted based on the y-axis.

can take on integer or real values. The optimization criteria is specified as a weighted linear combination over a subset of the variables in the problem that should either be maximized or minimized. We do not need to go into further detail, as the MILP model presented in this section is quite basic and uses only integer variables for the encoding.

Here, we present a version of the MILP model introduced by Do and Kambhampati for comparison to our partial weighted MaxSAT model. The modifications fall under three categories: (1) fixes for bugs in the original formulation, (2) removal of variables and constraints not relevant to our setting (i.e., the temporal and resource related portions of the model), and (3) adding constraints to enforce that a solution is the transitive closure. The variables we use for the model include the following.

$$X^f_{a_j,a_i} = \begin{cases} 1 & \text{when } a_i \text{ supports } a_j \text{ with fluent } f \\ 0 & \text{otherwise} \end{cases}$$

$$Y^f_{a_i,a_j} = \begin{cases} 1 & \text{when } a_i \text{ is ordered before } a_j \text{ due to interference on fluent } f \\ 0 & \text{otherwise} \end{cases}$$

$$O_{a_i,a_j} = \begin{cases} 1 & \text{when } a_i \text{ is ordered before } a_j \\ 0 & \text{otherwise} \end{cases}$$

Note that $X^f_{a_j,a_i}$ and O_{a_i,a_j} are analogous to $\Upsilon(a_i, f, a_j)$ and $\kappa(a_i, a_j)$ respectively. The interference variables $Y^f_{a_i,a_j}$ are defined only for those cases where a_j can conflict with the execution of a_i on fluent f: either $f \in (PRE(a_i) \cup ADD(a_i)) \cap DEL(a_j)$ or $f \in (PRE(a_j) \cup ADD(a_j)) \cap DEL(a_i)$ holds. The constraints for the MILP model are as follows (unbound variables are assumed to be universally quantified).

- Interfering actions must be ordered (defined only for pairs of actions that interfere):

$$Y^f_{a_i,a_j} + Y^f_{a_j,a_i} = 1$$

- Every precondition is supported exactly one way:[8]

$$\forall f \in PRE(a_j), \quad \sum_{a_i \in adders(f)} X^f_{a_j,a_i} = 1$$

- Every support is threat free:

$$\forall a_d \in deleters(f), \quad (1 - X^f_{a_j,a_i}) + (Y^f_{a_d,a_i} + Y^f_{a_j,a_d}) \geq 1$$

- Support implies ordering:

$$O_{a_i,a_j} - X^f_{a_j,a_i} \geq 0$$

[8]The original paper had this constraint erroneously listed as $\sum_{a_i \in adders(f)} X^f_{a_i,a_j} = 1$.

- Interference implies ordering:

$$O_{a_i,a_j} - Y^f_{a_i,a_j} \geq 0$$

- Enforce the transitive closure of ordering constraints:

$$(1 - O_{a_i,a_j}) + (1 - O_{a_j,a_k}) + O_{a_i,a_k} \geq 1$$

- Forbid self loops in the ordering:

$$O_{a,a} = 0$$

- Order everything after the initial state action and before the goal action:

$$O_{a_I,a} = 1$$
$$O_{a,a_G} = 1$$

The final three constraints do not appear in the original model. The last one replaces constraints that referenced temporal variables to achieve the same effect, and the first two ensure that a solution is transitively closed. Not including these constraints would amount to optimizing over the transitive reduction, and as pointed out by Bäckström [4], minimizing the transitive closure has a greater appeal because it leads to fewer long chains in the plan. Finally, the optimization criteria for the MILP model is as follows.

$$Minimize \sum_{a_1,a_2 \in \mathcal{A}} O_{a_1,a_2}$$

The above model will produce reorderings of the input plan as feasible solutions, and will find a minimum reordering if solved to optimality. We implemented the MILP model using the state-of-the-art MILP solver Gurobi (version 5.0.2) [46], and measured the coverage over all domains as a function of time. Figure 3.4 contains the results.

We found that using the MILP model was effective for the smaller problems (those solved in under 2 seconds), but for anything more difficult, solving the partial weighted MaxSAT encoding with Sat4j proved more efficient. Overall, 101 problems were solved using Sat4j on the partial weighted MaxSAT encoding while only 87 problems were solved using Gurobi on the MILP model.

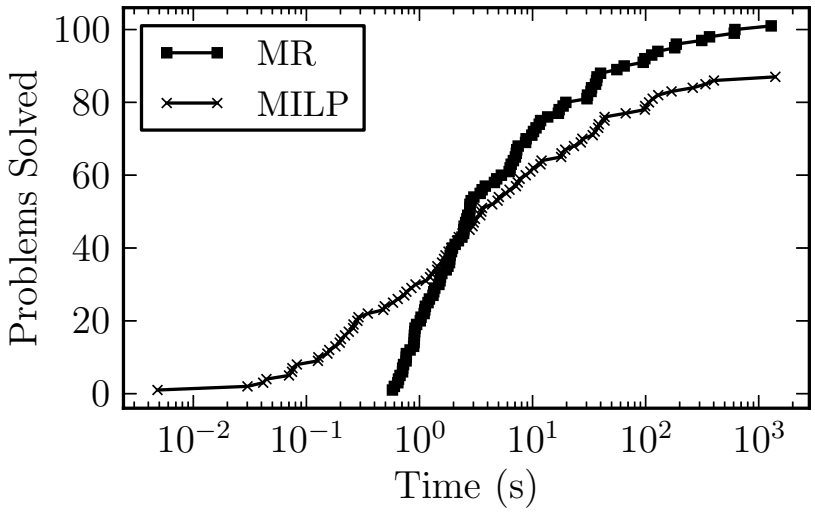


Figure 3.4: For a given timeout (x-axis), the number of problems solved within that timeout bound (y-axis) by (1) Sat4j using the MR encoding and (2) Gurobi using the MILP encoding described in the text.

The MILP model could, in theory, be modified to accommodate for the MCLCP criteria. However, given the results demonstrated in Figure 3.4, we suspect that using the MILP model would not provide an improvement over using our partial weighted MaxSAT encoding with Sat4j.

3.7 Discussion

In this chapter, we proposed a practical method for computing the optimal deordering and reordering of a sequential or partial-order plan. Despite the theoretical complexity of computing the optimal deordering or reordering being NP-hard, we are able to compute the optimal solution by leveraging the power of modern MaxSAT solvers. We further proposed an extension to the classical least commitment criteria of minimal deordering and reordering: a minimum cost least commitment POP (MCLCP). An MCLCP considers the total cost of actions in a solution before minimizing the number of ordering constraints. Central to the encodings we propose is a notion of ordering relevance: we designed the optimization criteria to minimize the ordering constraints in the resulting plan, leaving only those that are relevant for plan validity.

Our approach uses a family of novel encodings for partial weighted MaxSAT, where a solution corresponds to an optimal POP satisfying one of the three least commitment criteria we investigate: minimum deordering, minimum reordering, and our proposed

minimum cost least commitment POP. We solve the encoding with a state-of-the-art partial weighted MaxSAT solver, Sat4j, and find that the majority of problems are readily handled by the MaxSAT solver in a reasonable amount of time.

We also investigated an existing polynomial algorithm for deordering sequential plans and discovered that it successfully computes the optimal deordering in every problem we tested, despite its lack of theoretical guarantee. Because the algorithm is fast in practice, it is well suited for relaxing a POP if we require a deordering.

Here, we discuss work related to ours and conclude with a discussion of potential future work.

3.7.1 Related Work

The standard SAT-based planning encodings also produce a POP (e.g., Kautz *et al.* [54]), but a significant difference between the standard encodings and our work is that we avoid encoding an action in every layer in a planning graph by appealing to the fact that we already know the (superset of) actions in the solution. Intuitively, we can view the encoding as using MaxSAT to find the implicit layers for the actions in our plan by way of computing the relevant ordering constraints. An additional difference is that choosing a layer for every action unnecessarily restricts the timing of that action when it can potentially appear in multiple adjacent layers.

The core of our encoding is similar to the causal encodings of Kautz *et al.* [54] and Variant-II of Robinson *et al.* [88]. We similarly encode the ordering between any pair of actions as a variable ($\kappa(a_i, a_j)$ in our case), but rather than encoding every potential action occurrence or modelling a relaxed planning graph, we encode the formulae that must hold for a valid POP on the specific set of actions provided as part of the input. As mentioned in Section 3.2.3, there are also similarities between our work and that of Do and Kambhampati [29]. In particular, the optimization criteria for minimizing the number of ordering constraints coincides, as does the optional use of constraints to force a deordering. While Do and Kambhampati focus on temporal relaxation in the context of action ordering, we take the orthogonal view of minimizing the total action cost.

Finally, the work of Hickmott and Sardina focuses on generating partial-order plans through Petri net unfolding [50]. Of particular interest, they point out that a theoretical property of their approach is that the resulting plan will be a minimal deordering that respects *strong independence*. The notion of strong independence is a restriction on the unordered actions in the partial-order plan: there can be no ambiguity in which action produces a particular fluent. As a result, if a pair of actions both produce the same

fluent f, they can only be unordered if neither are required to produce f for the goal or some other action in the plan. This restriction makes the deorderings produced by the unfolding more restricted than the optimal deorderings produced by our approach. Another significant difference is that they are solving the planning problem from scratch rather than finding a deordering of an existing plan.

3.7.2 Conclusion

The use of our method for computing optimally relaxed plans provides two key advantages: (1) the optimal deordering provides a useful baseline for demonstrating the effectiveness of the Relaxer Algorithm, and (2) if minimizing action costs and/or maximizing flexibility is large concern, then solving the encoding can lead to far more flexible solutions than the Relaxer Algorithm can achieve. Our work leaves open the possibility for a heuristic approach similar to the Relaxer Algorithm that is capable of producing reorderings of a POP.

One further extension of our work is to consider different forms of optimization criteria. For example, one may change the soft clauses so as to minimize the number of fluents from the initial state that are required for plan validity. Doing so has the potential to improve planning formalisms that attempt to minimize the reliance on information about the initial state, such as assumption based planning [26].

Chapter 4

Robustly Executing Partial-Order Plans

4.1 Introduction

In the previous chapter, we presented a method for optimally relaxing a plan to be a maximally flexible POP. The motivation for doing this was to improve the robustness of plan execution. With the same motivation in mind, in this chapter we extend the typical POP execution strategy to take advantage of the flexibility inherent in a POP. The advantage of a partial-order plan over a sequential one is that the ordering of some actions can be chosen by the agent at runtime. Here, we improve the robustness further by allowing the agent to arbitrarily choose to execute any plan fragment of the POP at each step.

Unlike a sequential plan that specifies a set of actions and a total order over those actions, a POP only specifies those action orderings necessary for the achievement of the goal. In so doing, a POP embodies a family of sequential plans – a set of linearizations all sharing the same actions, but differing with respect to the order of those actions. Traditionally, an agent is left to choose any one of the linearizations during execution. By focusing on what is relevant for some portion of a linearization to achieve the goal, our work enables an agent to dynamically switch between executing different linearizations at run time. Further, the agent will always choose the linearization that achieves the goal with the lowest total cost.

An agent's model of the world is often imperfect and incomplete, which can lead to changes in the state of the world that deviate from the state the agent predicted. In execution monitoring (EM), an agent monitors the state of the world as a plan is being

executed. When there is a discrepancy between the predicted and observed state of the world, a typical EM system attempts to repair the plan or replan from scratch. EM approaches typically have many stages including state estimation, evaluating if execution can continue (state evaluation), selecting the action to perform (action selection), and replanning in the presence of plan failure. We are primarily concerned with the stages *state evaluation* and *action selection*. Given a plan and the current state of the world, can the agent continue executing the plan, and if so how should it proceed.

Effective EM systems determine the subset of *relevant* conditions that preserve plan validity and focus on discrepancies with respect to these conditions. Shakey the robot's triangle tables were an approach to modeling such conditions [34]. Recently, Fritz and McIlraith characterized the conditions under which a partially executed sequential plan remains valid as the regression of the goal back through the plan that remains [38]. An EM algorithm compares the conditions associated with each step of the plan to the state of the world and proceeds with the plan from the matching point. For partial-order planning, EM is perhaps best exemplified by the SIPE [100] and Prodigy [96] systems which take a different approach: monitoring the violation of so-called *causal links* and attempting to repair them as necessary.

In this chapter we address the problem of POP EM. Similar to the insights used for sequential plan EM, we appeal to a notion of logical regression to characterize the relevant conditions under which a POP remains viable. We then compile these conditions into a structured policy that compactly maps them to an appropriate action. The policy allows for robust and nimble execution by the agent, and we support this claim with both theoretical properties of the generated policy and empirical evaluations of the overall system. Figure 4.1 shows an overview of our framework for POP EM.

In the first phase, we create an initial POP from an input domain and problem file. There are a number of options for this step, but for simplicity we use the Relaxer Algorithm described in Chapter 3. In the second phase, where we make most of the contributions in this chapter, we generalize the POP to a list of condition-action tuples and subsequently create a structured policy that maps the state of the world to an appropriate action. The final phase uses the previously created policy for online execution, and this phase is where the majority of our experimental evaluation takes place. In the presence of unhandled states of the world, it is natural to pose new problems to the entire process. While we do not address this aspect in our work, replanning is a method that could further improve the policy's robustness.

We evaluate our approach analytically and empirically, comparing it to the standard technique for monitoring sequential plans. On International Planning Competition (IPC)

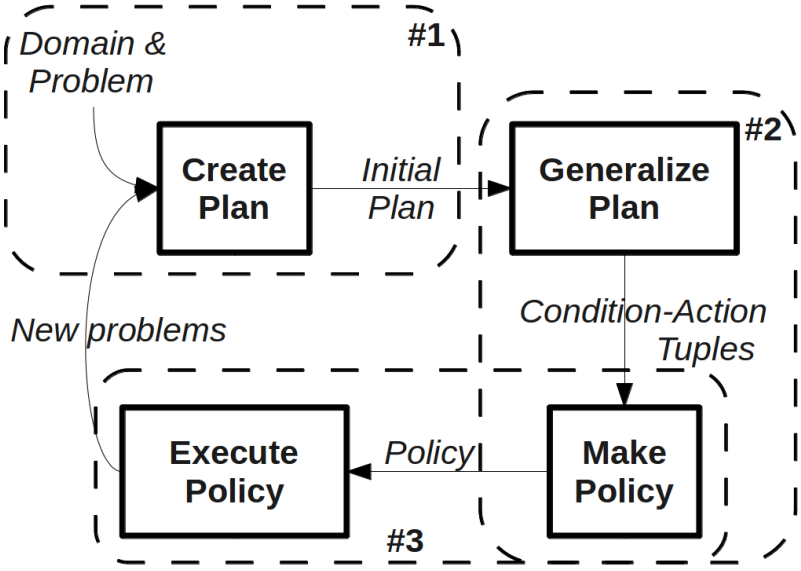


Figure 4.1: Overview of the approach for EM of a POP.

domains where there are numerous ordering constraints, and as such few distinct POP linearizations, our approach is 2 to 17 times faster and up to 2.5 times more robust. On POPs that have few ordering constraints, the number of states for which our POP remains viable may be exponentially larger than the number of such states resulting from using a sequential plan. We also identify further characteristics of a POP that give our approach a distinct advantage.

4.1.1 Contributions

In this chapter, we make the following main contributions:

- We provide a characterization of what makes a POP viable – the conditions under which some linearization can achieve the goal from a given state. Intuitively, a POP is viable if at least one linearization can achieve the goal, and it is valid if every linearization can achieve the goal. The benefit of considering viability of a POP stems from the fact that we should continue executing as long as there is still a way to accomplish our task.

- We develop a procedure for efficiently regressing a formula through a POP. We compute the minimal set of conditions for POP viability by performing logical regression implicitly over the POP's set of linearizations. Our technique leverages the commonality between linearizations to perform the computation efficiently, even in the presence of exponentially many linearizations.

- We present a means for generalizing a POP into a list of condition-action tuples. The condition indicates the viability of a fragment of the POP: a linearization of the fragment starting with the action in the tuple can achieve the goal as long as the current state satisfies the condition. The order of tuples in the list corresponds to the quality of the fragment from which they are generated, giving us a means of selecting the best tuple appropriate for a given state.

- Given the list of condition-action tuples, we present a method for combining them all into a single policy that is capable of checking every tuple simultaneously. Given a state of the world, the policy will return an action at the start of the smallest plan fragment that can achieve the goal. The use of a policy, in lieu of the list of tuples, provides a drastic improvement in the time required to find the appropriate action.

- We prove a number of theoretical properties of our methods including (1) the soundness and completeness of our algorithm to compute POP viability, (2) a limitation to the use of POP viability in the context of optimal input plans, and (3) the key property of the policy we generate to return the best action for the current situation.

- Taking advantage of our choice in policy representation, we provide an analytical means for computing the number of states for which our policy is applicable. Having the precise number of states allows us to directly measure the robustness of our generalized plan. The analytical result further allows us to establish a clear correlation between the flexibility of a POP and the robustness of its generalization.

- We identify the characteristics of a POP that maximize flexibility. One key property of a POP that leads to an increase of flexibility is the presence of multiple achievers of a fluent when the achievers are unordered with an action that requires the fluent. Ultimately, the flexibility of a POP depends not only on the number of linearizations it encompasses, but additionally on the number of ways that key fluents can be made true.

The work in this chapter is based on a publication at IJCAI'11 [74]. The largest change from the publication is the inclusion of the characterization of our generalized plan, which can be found in Section 4.5. Further, in this chapter we have expanded on the theoretical contributions of the publication.

4.1.2 Organization

We start this chapter with the required background material in Section 4.2 and follow with the formal characterization of state relevance in Section 4.3. In Section 4.4 we describe our characterization of POP viability. Next, we discuss our generalized plan representation in Section 4.5. We present an evaluation of our techniques in Section 4.6 and conclude with a discussion of related work and summary in Section 4.7.

4.2 Preliminaries

In this section, we review the notation required for our approach to generalizing and executing partial-order plans. We primarily draw upon the techniques of ordered decision diagrams to construct and represent the policy our agent should follow, and we generalize the techniques from sequential plan monitoring to work for partial-order plans.

4.2.1 Decision Diagram Operations

To represent the policy for execution, we use an Ordered Algebraic Decision Diagram (OADD) [5] where the inner nodes correspond to the truth of a fluent and the leaf nodes correspond to actions or the specially designated $\perp$ symbol which signifies that the agent does not know what to do. We will also make use of Ordered Binary Decision Diagrams (OBDDs) that use fluents as inner nodes, and we assume that every OADD and OBDD will have the same ordering of inner nodes (i.e., we have a total ordering of the fluents in our planning problem).

One operation that is central to our approach is the If-Then-Else (ITE) operator for two OADDs and an OBDD [5]. The ITE operator combines two OADDs (Pol_T, Pol_E) using an OBDD (Pol_I) to produce an OADD with the following semantics:

$$ITE(Pol_I, Pol_T, Pol_E) = \begin{cases} Pol_T(s) & \text{if } Pol_I(s) \\ Pol_E(s) & \text{if } \neg Pol_I(s) \end{cases}$$

As a building block to our approach, we need the ability to create a *primitive OADD* that returns an action a if and only if a set of fluents ψ holds in a state s. To create a policy for the pair (ψ, a), we follow the pre-defined ordering of fluent variables that respects ψ until it is fully implied in the OADD, and then add a as a leaf node.

For example, assume our total ordering of fluents is $p_1, \ldots, p_5$, and we want to create the primitive OADD for the pair $(\{p_2, p_4\}, a_1)$. The result is shown in Figure 4.2. Notice

that the ordering of inner nodes from the root to the leaf follows the fixed ordering. We use **policy**(ψ, a) to denote the primitive OADD that we construct in this manner.

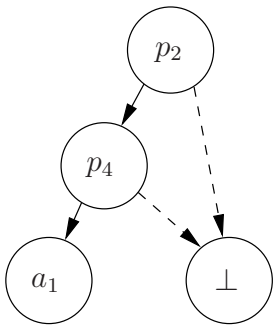


Figure 4.2: Primitive OADD for the pair $(\{p_2, p_4\}, a_1)$ under the ordering $p_1, \ldots, p_5$.

Finally, we will use the notation *obdd*(Pol) to denote the conversion of an OADD Pol into an OBDD. To convert an OADD to a corresponding OBDD, we simply replace every leaf node that contains an action with $\top$.

4.2.2 Sequential Plan Execution Monitoring

We take an approach to execution monitoring (EM) that builds directly on insights from EM for sequential planning dating back as far as Shakey the robot [34], and recently formalized by Fritz and McIlraith [38]. A central task of execution monitoring is to determine whether the plan being executed remains valid, given what is known about the current state. Recall that given a planning problem $\Pi = \langle F, O, I, G \rangle$, a sequential plan is *valid* if and only if the plan is executable in I and G holds in the resulting state.

We extend this definition to say that the sequential plan *remains valid* with respect to s and G if and only if there is a suffix of the plan that is executable in s and G holds in the state resulting from executing the suffix in s. It is the validity with respect to the current state that is at the core of monitoring sequential plan execution.

Algorithm 5 is a reproduction of the EM algorithm for sequential plans presented by Fritz and McIlraith [38] (cf. "Algorithm for Monitoring Plan Validity"). Line 2 can actually be improved by caching the regression results for increasing suffix size, and we use this technique in our implementation. Generally, the approach looks at increasingly larger suffixes of a sequential plan until the smallest suffix that can achieve the goal from s is found. If no such suffix exists, then $\perp$ is returned. We refer to this approach as the *Sequential Method*.

Algorithm 5: Sequential Method

Input: Problem $\langle F, O, I, G\rangle$, sequential plan $\vec{a} = [a_1, \cdots, a_n]$, and current state s.
Output: Action to execute a, or $\perp$ if $\vec{a}$ does not remain valid.

```
for i = n to 1 do
    if s ⊨ R*(G, [a_i, ··· , a_n]) then
        return a_i;
return ⊥;
```

4.3 State Relevance

The key insight behind our approach is to focus only on what is relevant for the agent to achieve the goal. A POP naturally represents a family of strategies for achieving the goal from a single initial state, but through careful generalization of the plan we are able to produce a far more flexible solution. When the world changes in unexpected ways, certain actions in a plan may no longer be required or previously executed actions must happen again. We can address this unpredictability by determining what is relevant in a given state for some partial plan to achieve the goal.

The idea of state relevance transcends simply determining if a partial plan can achieve a goal objective. As we will see in Chapter 6, it can also be applied to great effect in other situations. To capture the notion formally, we define state relevance as follows:

Definition 7 (State Relevance)**.** Given a planning problem $\Pi = \langle F, O, I, G\rangle$ with associated state space $\mathcal{S}$ over the fluents F and a boolean property Φ that represents some function over $\mathcal{S}$, the partial state $p \subseteq F$ is a *relevant partial state* with respect to Π and Φ iff for every state $s \in \mathcal{S}$, $s \models p$ implies that $\Phi(s)$ evaluates to true.

The definition of state relevance is intentionally made to be general; the notion will appear multiple times throughout the remainder of this dissertation. Some examples of the property Φ that we will consider are the following:[1]

- (cf. Section 4.4) "The goal G holds after executing $\vec{a}$ in state s.":

$$\Phi(s) \quad \textit{iff} \quad s \models \mathcal{R}^*(G, \vec{a})$$

- (cf. Section 6.4) "The goal G cannot be achieved from state s.":

$$\Phi(s) \quad \textit{iff} \quad \nexists \vec{a}, \mathcal{P}^*(s, \vec{a}) \models G$$

[1]The notation used in the examples is specific to the situation, and will be introduced as needed.

- (cf. Section 6.3) "The policy Pol will know what action to perform in state s.":

$$\Phi(s) \quad \textit{iff} \quad Pol(s) \neq \bot$$

The notion of state relevance has existed in the literature for quite some time: the first example we provided above – the subset of a state that is sufficient for ensuring a sequence of actions can achieve the goal – is at the very core of the early work in logical regression (e.g., Fikes *et al.* [34]). Similarly, Kobolov *et al.* [60] introduce the notion of a "basis function" – a relevant partial state sufficient for a sequence of actions to reach the goal with non-zero probability in a non-deterministic setting. In a more general sense, Nebel *et al.* [79] formally define various notions of relevance and focus primarily on computing a *minimal set of initial facts*, which we can view as a notion of state relevance that pertains only to the initial state. Finally, in a more direct sense, state relevance plays a role in the majority of techniques that use a decision diagram to map the state of the world to some other domain (e.g., Boolean values, actions, etc.). Examples of this include the match tree representation in many modern planners [48] and the policy representation for the non-deterministic planner, Gamer [58].

4.4 POP Viability

In Section 2.2.2 we defined the notion of POP validity intuitively as the property of a POP where every linearization achieves the goal from the initial state. With the ultimate objective being to achieve the goal, POP validity is in a sense too strong for what we desire: *some* way for the agent to reach the goal. In this section, we define such a notion – *POP viability* – and discuss how we can efficiently compute it using state relevance. We additionally provide the theoretical properties for POP viability.

4.4.1 Characterization

We could define the notion of a POP remaining valid with respect to a state and a goal, just as we have done for a sequential plan, but the validity condition is clearly too strong. Given that a POP compactly represents multiple linearizations, an appropriate analogue is to ensure that at least one of these linearizations remains valid. With this aim, we define the notion of POP viability formally as follows:

Definition 8 (POP viability)**.** Given a planning problem $\Pi = \langle F, O, I, G \rangle$, POP P, and state s, P is *viable* with respect to state s and G iff there exists a linearization of P that

is valid with respect to s and G.

Intuitively, a POP P is viable with respect to s and G whenever there exists a linearization of P that has a suffix $\vec{a}_{suf}$ that is a valid plan for the problem $\langle F, O, s, G\rangle$. Whereas the objective of EM for sequential plans is to determine whether a plan remains valid, we claim that the objective of POP EM is to determine if the POP is viable with respect to the current state.

Following the methodology adopted for EM of sequential plans, we can address this question effectively by identifying the relevant conditions that lead to POP viability and ensure that one of these conditions holds. In the next section, we present a method for computing every such suffix $\vec{a}_{suf}$, along with the sufficient condition that s must satisfy for $\vec{a}_{suf}$ to achieve the goal.

4.4.2 Condition-Action List

As there may be exponentially many linearizations of a POP, computing the conditions for each one is inefficient. There is often structure in a POP, however, that we can exploit to compute the conditions for POP viability more efficiently. To this end, we provide a method for constructing conditions that avoids enumerating all of the linearizations. Intuitively, we regress the goal back through the POP, exploiting the conditions and actions shared amongst the linearization suffixes. During the process, we gradually reduce the POP until we have enumerated every condition.

Our general approach is to enumerate the viability conditions for a unique partially executed POP: a *POP Suffix* (cf. Section 2.2.2). For a POP suffix P' of the POP P, every linearization of P' will correspond to the suffix of a linearization of P. We can define the notion of *POP Prefix* analogously by insisting $a_I \in \mathcal{A}'$ (in lieu of a_G) and replacing the final condition with the following:

$$\text{if} \quad \forall a_1 \in \mathcal{A}, \forall a_2 \in \mathcal{A}', \ (a_1 \prec a_2) \in \mathcal{O} \quad \text{then} \quad a_1 \in \mathcal{A}'$$

To construct the POP viability conditions we use the following notation:

- $last(\langle \mathcal{A}, \mathcal{O}\rangle)$: The actions that appear in a POP such that there is no ordering constraint originating from the action (i.e., it appears at the end of the POP):

$$last(\langle \mathcal{A}, \mathcal{O}\rangle) \stackrel{\text{def}}{=} \{a \mid a \in \mathcal{A} \wedge \nexists\, (a \prec a') \in \mathcal{O}\}$$

- $prefix(\langle \mathcal{A}, \mathcal{O}\rangle, a)$: The POP that remains after we remove action a and all of the

associated ordering constraints from the POP.

$$prefix(\langle \mathcal{A}, \mathcal{O} \rangle, a) \stackrel{\text{def}}{=} \begin{cases} \langle \mathcal{A} \backslash a, \mathcal{O} - \{(a' \prec a) \mid a' \in \mathcal{A}\} \rangle & \text{if } a \in last(\langle \mathcal{A}, \mathcal{O} \rangle) \\ undefined & \text{otherwise} \end{cases}$$

The $prefix$ operation essentially slices one action off the end of a POP, giving us a new POP. Note that if defined, $prefix(\langle \mathcal{A}, \mathcal{O} \rangle, a)$ is a POP prefix of $\langle \mathcal{A}, \mathcal{O} \rangle$. We now have the notation required to define the set of conditions for POP viability:

Definition 9 (Γ-conditions)**.** A Γ-condition is a tuple containing a formula and a POP. Given a planning problem $\Pi = \langle F, O, I, G \rangle$ and POP $P = \langle \mathcal{A}, \mathcal{O} \rangle$, we define the set of Γ-conditions for Π and P as $\Gamma_{\Pi,P} = \bigcup_{i=0}^{|\mathcal{A}|} \gamma_i$, where $\gamma_0 = \{\langle G, P \rangle\}$ and we define γ_i inductively as follows:

$$\gamma_{i+1} = \bigcup_{\langle \psi, P \rangle \in \gamma_i} \{\langle \mathcal{R}(\psi, a), prefix(P, a) \rangle \mid a \in last(P)\} \tag{4.1}$$

Every tuple in γ_i contains a condition for a linearization suffix of size i to be a valid plan from a state matching the condition, as well as a POP that contains the actions not in the suffix (i.e., a corresponding POP prefix). Later, we formally relate the conditions for POP viability and the formula $\Gamma_{\Pi,P}$ in Theorem 5.

With a method for computing the required conditions for POP viability, we turn our attention to how we exploit these conditions for the overall EM strategy. To put the conditions for POP viability to use, we must determine what the agent's behaviour should be when a condition is met. Below we deal with the case when the current state satisfies more than one condition, but assuming that a condition is met, we ultimately want to return an appropriate action.

In the construction of the Γ-conditions, we continually choose the next action through $last(P)$. To build a mapping of conditions to actions, we record the action that was used to construct a condition in an ordered list called the *Condition-Action List.* Our final condition-action list will be made up of tuples that contain a regressed formula, a single action, and a cost for the tuple.[2] Using the construction of $\Gamma_{\Pi,P}$, we present a procedure for computing the condition-action list in Algorithm 6. The special notation

[2]A tuple is of higher quality if it has lower cost.

SuffixCost$(P, \mathcal{A})$ denotes the total cost of actions not found in the POP P:

$$\textsc{SuffixCost}(\langle \mathcal{A}', \mathcal{O}' \rangle, \mathcal{A}) = \sum_{a \in \mathcal{A} - \mathcal{A}'} c(a)$$

Algorithm 6: Condition-Action List Generator

Input: POP $\langle \mathcal{A}, \mathcal{O} \rangle$. Planning problem $\Pi = \langle F, O, I, G \rangle$.
Output: List of (ψ, a, c) tuples.

```
1 L = [ ];   // L is the list of (ψ, a, c) tuples to be returned
2 Γ = {⟨G, ⟨A, O⟩⟩};   // Γ is a set of tuples of the form ⟨ψ, P⟩
3 for i = 1 ··· |A| do
4     foreach ⟨ψ, P'⟩ ∈ Γ do
5         foreach a ∈ last(P') do
6             L.append( (R(ψ, a), a, SuffixCost(P', A) + c(a)) );
      /* Update to γ_{i+1}                                          */
7     Γ = ⋃_{⟨ψ,P'⟩∈Γ} {⟨R(ψ, a), prefix(P', a)⟩ | a ∈ last(P')};
8 L.sort(c-value);   // Sort L based on the c-value of the tuple
9 return L;
```

The algorithm begins by initializing Γ to contain the entire POP, and the goal as the associated formula. In each iteration, we update Γ and add the (ψ, a, c) tuples to the list. Note the order of the (ψ, a, c) tuples in L: if one tuple appears after another, we know its cost must be equal or greater. The ordering of L is crucial for our approach, as we prefer to execute actions to achieve the goal while minimizing the total action cost.

We now have a specification of our objective (POP viability), and an algorithm to compute the conditions under which a POP remains viable (the condition-action list computed by Algorithm 6). Next, we look at the theoretical properties of these concepts.

4.4.3 Theoretical Results

Having motivated the desire for POP viability earlier in this section, as opposed to POP validity, we turn to the theoretical results that surround our approach for computing the former. We have already described the connection between the suffix of a POP's linearization and the POP's viability. To build on the result, we state it formally here.

Proposition 1. *Given a planning problem $\Pi = \langle F, O, I, G \rangle$, a POP P is viable with*

respect to state s and G iff at least one linearization of P has a suffix $\vec{a}_{suf}$ such that:

$$s \models \mathcal{R}^*(G, \vec{a}_{suf})$$

Proposition 1 follows immediately from the definition of when an action sequence "remains valid" and from Theorem 2. We can now draw a correspondence between the conditions we compute and POP viability:

Theorem 5 (Condition Correspondence)**.** *Given a planning problem $\Pi = \langle F, O, I, G\rangle$, the POP P is viable with respect to state s and G iff $\exists\ \langle \psi, P'\rangle \in \Gamma_{\Pi,P}$ such that $s \models \psi$.*

Proof. We begin by showing that:

(1) at least one linearization of P has a suffix $\vec{a}_{suf}$ such that $s \models \mathcal{R}^*(G, \vec{a}_{suf})$

iff

(2) there exists a pair $\langle \psi, P'\rangle$ in $\Gamma_{\Pi,P}$ such that $s \models \psi$.

We do so by showing a correspondence between the linearization suffixes that exist, and the tuples in $\Gamma_{\Pi,P}$: for every linearization suffix of P, $\vec{a}_{suf} = [a_k, \ldots, a_n]$, there exists a POP P' such that $\langle \mathcal{R}^*(G, \vec{a}_{suf}), P'\rangle \in \gamma_{n-k+1}$. For the base case, we have the empty suffix and the trivial initial condition in γ_0 suffices: $\langle \mathcal{R}^*(G, [\,]), P\rangle = \langle G, P\rangle \in \gamma_0$.

Inductively, we assume that for a given $i = n-k+1$, for every linearization suffix of P, $\vec{a}_{suf} = [a_k, \ldots, a_n]$, there exists a POP P' such that $\langle \mathcal{R}^*(G, \vec{a}_{suf}), P'\rangle \in \gamma_i$. Without loss of generality, consider any suffix of P, $[a_{k-1}, a_k, \ldots, a_n]$. By the inductive hypothesis, we know that there exists a POP P' such that $\langle \mathcal{R}^*(G, [a_k, \ldots, a_n]), P'\rangle \in \gamma_i$. Following equation 4.1, we know that γ_{i+1} will contain $\langle \mathcal{R}(\mathcal{R}^*(G, [a_k, \ldots, a_n]), a_{k-1}), prefix(P', a_{k-1})\rangle$. Note that a_{k-1} must be in $last(P')$, otherwise $[a_{k-1}, a_k, \ldots, a_n]$ would not be a linearization suffix. Because $\mathcal{R}(\mathcal{R}^*(G, [a_k, \ldots, a_n]), a_{k-1}) = \mathcal{R}^*(G, [a_{k-1}, a_k, \ldots, a_n])$, we can conclude that every linearization suffix has a corresponding tuple in $\Gamma_{\Pi,P}$.

If every linearization suffix $\vec{a}_{suf}$ of the POP P has a corresponding pair $\langle \mathcal{R}^*(G, \vec{a}_{suf}), P'\rangle$ in $\Gamma_{\Pi,P}$, then we can conclude that forward direction of our original claim must hold (i.e., (1) $\rightarrow$ (2)). For the opposite direction, note that the pairs in $\Gamma_{\Pi,P}$ are constructed by enumerating suffixes of increasing size – the goal condition is regressed through actions at the end of the POP that are subsequently removed. If there existed a pair $\langle \psi, P'\rangle \in \Gamma_{\Pi,P}$ such that ψ did not equal the regression of the goal for any linearization suffix, then the sequence of actions that generated ψ must have violated the $a \in last(P)$ condition of equation 4.1: a contradiction.

Having shown that at least one linearization of P has a suffix $\vec{a}_{suf}$ such that $s \models \mathcal{R}^*(G, \vec{a}_{suf})$ iff $\exists \langle \psi, P' \rangle \in \Gamma_{\Pi,P}$ such that $s \models \psi$, we can conclude that Theorem 5 holds due to Proposition 1. □

While Proposition 1 provides us with the precise characterization of POP viability, Theorem 5 shows how the conditions for POP viability can be systematically computed. With a connection between POP viability and the set of conditions $\Gamma_{\Pi,P}$, we shift our focus to the correctness of our proposed algorithm for computing the conditions.

Theorem 6 (Soundness and Completeness of Algorithm 6)**.** *Given a planning problem* Π *and associated POP* P*, the conditions in the tuples returned by Algorithm 6, with input* P *and* Π*, have a one-to-one correspondence to the conditions in* $\Gamma_{\Pi,P}$*, and the associated actions correspond to the first action in the linearization suffix associated with the condition.*

Proof. The conditions computed by Algorithm 6 correspond precisely to those in $\Gamma_{\Pi,P}$ because line 7 performs the update for successive γ_i steps and line 6 adds the conditions for each step. Because the actions chosen in line 5 are from $last(P)$, the actions in the tuples correspond to the first action in the suffix associated with the condition. □

Given the completeness aspect of Theorem 6, we immediately have the following corollary that stipulates we can continue executing whenever it is possible to do so.

Corollary 1 (Execution Completeness)**.** *Given a planning problem* Π*, an associated POP* P*, and a state* s*, if some suffix of a linearization of* P *can achieve the goal from* s*, then there exists a condition* ψ *in the tuples returned by Algorithm 6 such that* $s \models \psi$*.*

4.5 Generalized Plan

Rather than execute a POP in the traditional manner, we can achieve far greater flexibility through a more generalized behaviour: continually execute the actions at the start of the best quality POP suffix that remains viable. We encompass this behaviour through the construction and use of a policy that maps the state of the world to an appropriate action. In this section, we characterize the generalized plan that we produce, describe the policy representation for the generalized plan, and outline the method we use to create it from a condition-action list. We additionally provide theoretical results that surround the representation.

4.5.1 Characterization

There are various notions of generalized planning [95], and our approach falls under the category of plans that are applicable in multiple initial states. The goal that our agent must achieve does not change, but the policy encompasses a family of plans that can achieve the goal from a variety of initial states. The formal characterization of the states from which our generalized plan can achieve the goal is as follows.

Property 1 (Generalized Plan Applicability)**.** Given a planning problem Π and a POP P, the set of states our generalized plan can reach the goal from consists of every state $s \in \mathcal{S}$ such that the following holds:

$$\exists \langle \psi, P \rangle \in \Gamma_{\Pi,P} \;\; s.t. \;\; s \models \psi$$

We do not insist that the input plan is optimal in any sense, but it is interesting to consider the effect of using the generalized plan that was produced from a cost-optimal plan. In this setting, a cost-optimal plan is a POP such that the total action cost is less than or equal to the total action cost of every other valid POP. In such a setting, the generalized plan does *not* provide any guarantee on the optimality of the plan it executes.

Theorem 7. *Given an optimal POP P for the planning problem Π, there may exist a state in which the generalized plan characterized by $\Gamma_{\Pi,P}$ can reach the goal from, but in doing so will be suboptimal.*

Proof. We prove Theorem 7 by way of an example. Consider the planning problem $\Pi = \langle F, O, I, G \rangle = \langle \{p, q\}, \{a_1, a_2\}, \{\}, \{p, q\} \rangle$ with the actions defined as:

$$\begin{array}{llll} PRE(a_1) = \{\} & ADD(a_1) = \{p, q\} & DEL(a_1) = \{\} & c(a_1) = 2 \\ PRE(a_2) = \{p\} & ADD(a_2) = \{q\} & DEL(a_2) = \{\} & c(a_2) = 1 \end{array}$$

The only plan for Π (and thus the optimal one) is $[a_1]$. The regression of the plan, however, covers every state: $\mathcal{R}(G, a_1) = \{\}$. Consider the state where p holds but q does not. The generalized plan will still achieve the goal, because $\{p\} \models \{\} = \mathcal{R}(G, a_1)$, but in doing so it will execute a_1 instead of a_2. This suboptimal course of action serves as an example of why the generalization of an optimal plan may not behave optimally. □

4.5.2 Representation

Our approach for execution monitoring of a POP is to generate a *structured policy* that maps states to actions. Given a state, the policy returns the action that gets us as close to the goal as possible. We refer to this procedure as the *POP Method.* By using the POP Method, we avoid the need to check numerous conditions for the current state. We also benefit from having an action returned that gets us as close to the goal as any linearization with the Sequential Method. Our contribution includes how we build, represent, and use the structured policy for execution monitoring. If we view the policy as a boolean function that maps to true whenever an action is returned by the policy, we can characterize the boolean function by the following formula (following Property 1):

$$\bigvee_{\langle \psi, P \rangle \in \Gamma_{\Pi,P}} \psi$$

For a state s, we define a *valid linear suffix* as a linearization suffix of our POP that is valid with respect to s. Using this notion, with a condition-action list we embody the following high-level behaviour in the agent's policy:

Property 2 (Opportunistic Property)**.** If at least one valid linear suffix exists, then return the first action of the valid linear suffix with the best quality. If more than one qualifies as having the best quality, pick one arbitrarily.

We achieve this property as long as the condition-action list is in the correct order: the order that corresponds to the condition-action list retuned by Algorithm 6. As for the ordering of fluents in the decision diagram, we found through experimentation that ordering them based on where they appear in the condition-action list typically produced a smaller policy when compared to other ordering heuristics.

4.5.3 Policy Construction

To build our policy, we apply the ITE method for OADD's in successive steps. Two key aspects for building the policy are how we choose the individual policies to begin with, and subsequently how we combine them into one overall policy. We do the former by creating a primitive policy for each (ψ, a, c) tuple in our condition-action list L, and we achieve the latter by repeated use of the ITE operation.

Using the notation introduced in Section 4.2, Algorithm 7 computes the overall policy. We use $L.pop()$ to indicate that the last element of L should be removed and returned. We initialize the overall policy π as the last tuple computed from Algorithm 6 (i.e., the

one furthest from the goal). At every iteration, we retrieve the next tuple from the condition-action list and combine it with π through the ITE operator.

Algorithm 7: POP Policy Generator

Input: Condition-Action List L in sorted order.
Output: Structured policy mapping state to action.

```
1 (ψ, a, c) = L.pop();    // Retrieve the initial tuple
2 π = policy(ψ, a);    // π is the current overall policy.
3 while |L| > 0 do
4 |   (ψ, a, c) = L.pop();    // Retrieve the next tuple
5 |   next = policy(ψ, a);
6 |   π = ITE(obdd(next), next, π);
7 return π;
```

It would be correct to build the final policy in the reverse order: converting the current overall policy to an OBDD at each step and combining the next primitive OADD from the reversed condition-action list. Doing so, however, leads to poor performance of the overall approach: the same final policy was constructed while taking far longer.

4.5.4 Theoretical Results

Having described the method for generating a policy, we turn our attention to proving that the policy satisfies the desired property.

Theorem 8. *The structured policy constructed by Algorithms 6 and 7 satisfies the opportunistic property.*

Proof. Without loss of generality, we assume that $a_1 \neq a_2$. Also, note that an action may appear in multiple tuples in L. Consider the case where there exists a state s such that $s \models \mathcal{R}^*(G, \vec{a}_1)$ and $s \models \mathcal{R}^*(G, \vec{a}_2)$, where $\vec{a}_1$ (resp. $\vec{a}_2$) is a valid linear suffix starting with a_1 (resp. a_2) as the action chosen in the construction of the condition-action list. Assume that $\vec{a}_2$ is *cheaper* than $\vec{a}_1$, and no cheaper valid linear suffix exists for the state s. Because Algorithm 6 adds a (ψ, a, c) tuple to L for every unique condition of a suffix at a particular size, both $(\mathcal{R}^*(G, \vec{a}_1), a_1, c(\vec{a}_1))$ and $(\mathcal{R}^*(G, \vec{a}_2), a_2, c(\vec{a}_2))$ will appear in L. Because the cost of $\vec{a}_2$ is cheaper than $\vec{a}_1$, $(\mathcal{R}^*(G, \vec{a}_2), a_2, c(\vec{a}_2))$ must appear before $(\mathcal{R}^*(G, \vec{a}_1), a_1, c(\vec{a}_1))$ in L. The semantics of ITE used on line 6 allows us to conclude that if $s \models \mathcal{R}^*(G, \vec{a}_2)$, then a_2 would be returned, not a_1. □

Theorem 8 is a powerful result that allows us to combine the simplicity of using a decision diagram with the power of returning the best action possible according to our scheme of execution. We empirically demonstrate this power in the following section.

4.6 Evaluation

We evaluate the claim that employing a POP and monitoring it using our POP Method can provide enhanced flexibility at execution time compared to the EM of a sequential plan using a standard EM method. To do so, we provide both an analytical and empirical evaluation of our approach compared to a standard approach for monitoring sequential plans; the Sequential Method (cf. Section 4.2). We use five domains from the International Planning Competition (IPC) to illustrate the advantage of using our approach: Depots, Driverlog, TPP, Rovers, and Zenotravel. We also investigate the features of a POP that lead to performance improvements through three expository domains: Parallel, Dependent, and Tail.

Experiments were conducted on a Linux desktop with a two-core 3.0GHz processor. Each run was limited to 30 minutes and 1GB of memory. Plans for the Sequential Method were generated by FF [51], and a corresponding POP for the POP Method was generated by using the Relaxer Algorithm (cf. Section 3.2).

4.6.1 Policy Efficiency

To measure the impact that using a policy can have for EM, we consider a POP that represents only one linearization. In such a case, the POP Method and Sequential Method will return the same action for any given state. Because we can use *any* valid POP for Algorithm 6, we can use the sequential plan, and then pass the resulting condition-action list to Algorithm 7 for the construction of the *Sequential Policy.* Because the Sequential Policy is able to query all conditions in the sequential plan with a single traversal of an OADD, the time required to return an action should be faster than the Sequential Method.

We refer to the *ratio of effort* as the total time for the Sequential Method to return a result for every state in a randomly generated set of 500 states, divided by the total time for the Sequential Policy to return the same actions. Figure 4.3 gives an indication of the time savings of our approach. Sorted based on the ratio of effort, the x-axis includes every problem from the five IPC domains. The y-axis indicates the ratio of effort for a given problem. For each problem, the same 500 random states were used for both

approaches. The Sequential Policy is 2 to 17 times faster, and the gains become more pronounced with larger plans.

With a mean ratio of 6, the use of a structured policy can have substantial gains when it comes to reacting quickly. While the absolute gains are small (on the order of milliseconds at times), the relative speedup may prove to be crucial for real-time applications such as RoboCup. In such domains, the agent must evaluate the state and decide on an action several times a second.

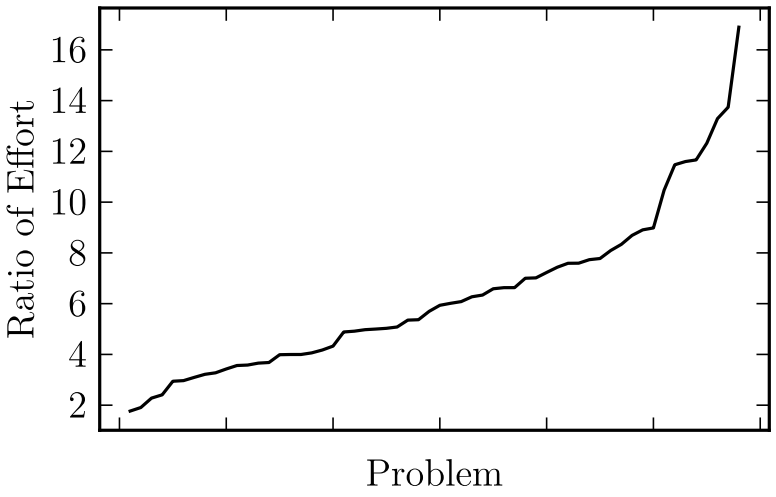


Figure 4.3: Efficiency of querying the structured policy. The y-axis indicates the total time for the Sequential Method (on 500 random states) divided by the total time for the Sequential Policy (on the same states). The x-axis ranges over all five IPC problem sets, and we sort the problems based on the y-axis value.

4.6.2 Analytical Results

In Section 4.1, we argued that a POP provides flexibility and robustness at run time. In this analysis we try to quantify the added flexibility afforded by the POP in concert with the POP Method, relative to the Sequential Method. We refer to the number of complete states for which an approach is capable of returning an action as the *state coverage*. We can measure the state coverage by using model counting procedures on the constructed OADD; a polynomial-time operation due to the ordered nature of OADDs [24]. In the case of the Sequential Method, we generate the OADD as in the previous section. The number of models for either OADD corresponds to the number of states for which the approach can return an action. Figure 4.4 shows the relative state coverage for the five IPC domains (the POP Method coverage divided by the Sequential Policy coverage).

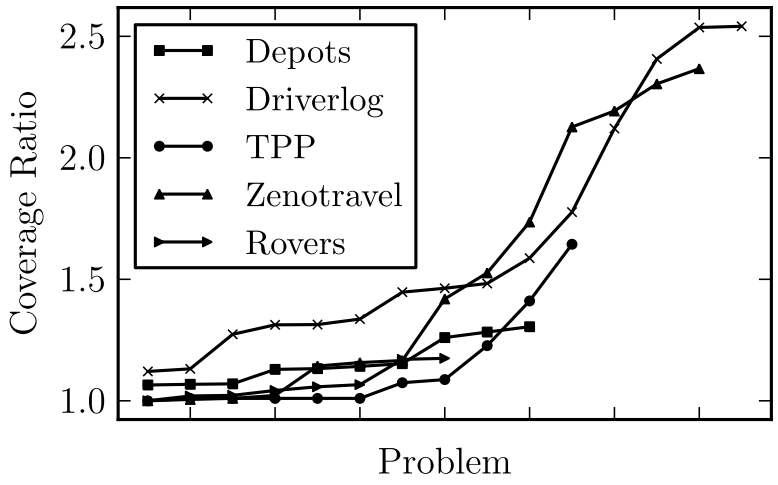


Figure 4.4: Analytic comparison of state coverage for the POP Method and the Sequential Method. The y-axis indicates the state coverage of the POP Method divided by the state coverage of the Sequential Method. We sort problems from each domain based on their y-axis value.

The y-axis indicates the ratio of states covered for a given problem: state coverage of the POP Method divided by the state coverage of the Sequential Method. For example, a value of 1.5 indicates that the POP Method returns an action in 50% more states than the Sequential Method. We sort problems from each domain based on their y-axis value.

The relative state coverage (or *coverage ratio*) ranges from 1 (i.e., the same number of states are handled) to 2.5. Further, we see a substantial increase in the state coverage for the Driverlog and Zenotravel domains in particular. Larger plans do not necessarily have a higher coverage ratio, and we conjecture that the ratio has more to do with the structure of a POP than its size. The state coverage is an approximation of a POP's flexibility, as the set of states used in the model count includes states that will never occur in practice, either because they are inconsistent or unreachable. Nonetheless, the coverage ratio gives us an approximate measure of the relative gain the flexibility of a POP has to offer when realized with our proposed approach.

4.6.3 Expository Domains

The evaluation above was performed on IPC domains which were designed to be challenging domains for sequential planning algorithms. As such, there tends to be significant dependencies between actions and the number of action orderings is large. The high level of dependency is not present in a variety of real-world planning applications (e.g., distributed plans for multiple agents). To evaluate our EM approach for POP, we investi-

gate key features we expect to find in real-world planning problems that are emphasized in three expository domains we designed: Parallel, Dependent, and Tail. Each domain provides a unique insight into the behaviour of POP Method.

Parallelism

The *Parallel* domain demonstrates the impact of multiple linearizations. We construct the Parallel domain so that a solution has k actions that can be performed in parallel. Each action has a single precondition satisfied by the initial state and a single add effect required by the goal. There are no ordering constraints among the actions, and so any permutation of the k actions is a valid plan. We show an example with $k = 3$ in Figure 4.5, and Appendix B.1 contains the full domain description.

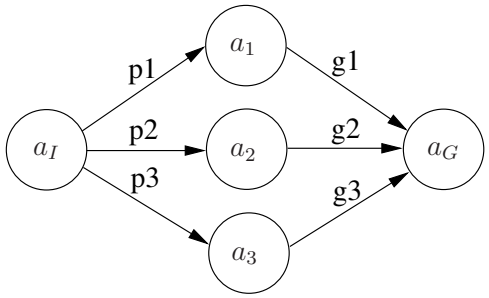


Figure 4.5: Example of the Parallel domain with $k = 3$.

As a consequence of there being no ordering constraints among the k actions, a solution to a problem in the Parallel domain has a large number of linearizations; with k parallel actions, there are $k!$ linearizations. If the actions mostly have different preconditions and effects, the POP Method will be applicable in many more states than the Sequential Method. There are many states that the Sequential Method fails to capture because of the limited number of unique conditions present in any single linearization, and every linear solution to a Parallel problem will have this property. In contrast, the POP Method captures the condition for every linearization suffix. Consequently, we find an exponentially increasing gap in state coverage.

The coverage ratio was computed for problems in the Parallel domain with k ranging from 2 to 10. We present the results in Figure 4.6. A clear exponential trend in the increase of state coverage occurs as we increase k.

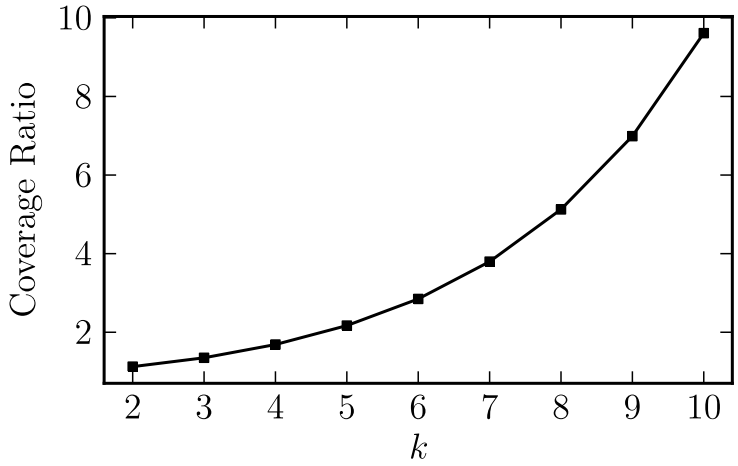


Figure 4.6: Analytic comparison of state coverage for the POP Method and the Sequential Method in the Parallel domain.

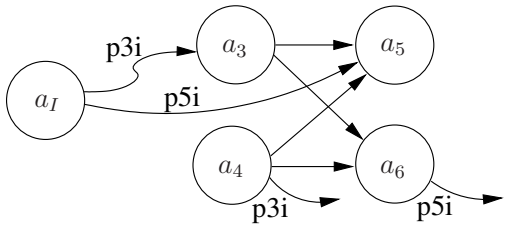


Figure 4.7: Excerpt from the POP of a Dependent domain problem. An edge with no endpoint signifies that the action has that fluent as an effect. We only label edges of interest.

Extra Support

An action has *extra support* if, for a precondition p, there are multiple actions in the POP that act as the achiever of p in at least one linearization. A POP is said to have extra support if one of its actions has extra support. We construct problems in the *Dependent* domain to require k successive pairs of actions such that one action ($a_\dashv$) has an extra precondition satisfied both by the initial state and the other action in the pair ($a_\vdash$).

An excerpt from a solution to a problem in the Dependent domain is shown in Figure 4.7, and the full domain description can be found in Appendix B.2. The initial state satisfies the precondition $p3i$ of action a_3. The precondition can also be satisfied by a_4 in any linearization that has a_4 ordered before a_3. If the dynamics of the world cause $p3i$ to become false during execution prior to executing a_3, then we must execute a_4 in order for a_3 to be executable.

The achiever of a fluent needed for extra support will depend on the linearization.

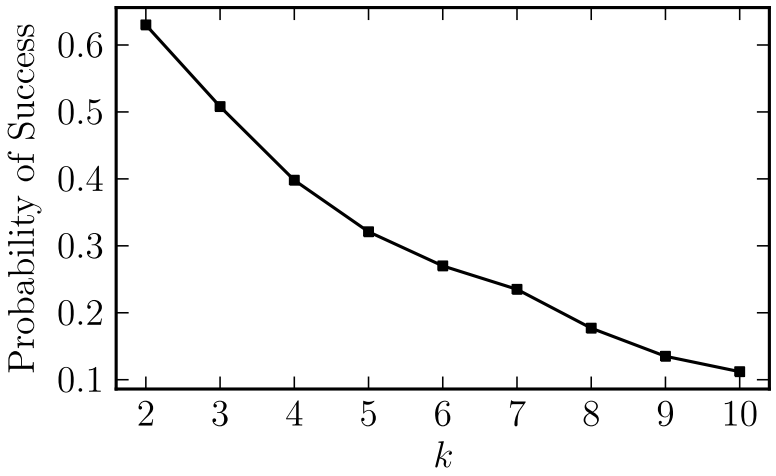


Figure 4.8: Success rate for reaching the goal for the Sequential Method in the Dependent domain (the POP Method always reaches the goal). The x-axis shows the problem size, and the y-axis shows the percentage of trials that reached the goal out of 1000.

Monitoring multiple linearizations, the POP Method is a more robust EM solution. Here we measure robustness by how likely an approach is to achieve the goal in a dynamic world. At every time step, we set a randomly selected fluent to false or do nothing if it is already false. We then query the approach and execute the action returned. The simulation repeats these two steps, and ends when either the current state satisfies the goal or the approach cannot return an action. We measure the likelihood of reaching the goal as the percentage of trials that end in the goal state.

For a problem in the Dependent domain with a given k, there are 2^k linearizations of the POP. Only one of these will have a 100% success rate when using the Sequential Method: the linearization that correctly orders every pair of actions so that $a_{\vdash}$ comes before $a_{\dashv}$. Using the default linearization generated by FF (which orders $a_{\vdash}$ *after* $a_{\dashv}$), we ran 1000 trials for both approaches. Figure 4.8 shows the result.

We can see that the likelihood of reaching the goal approaches zero for the Sequential Method. As the plans become longer, there is more opportunity for something to go wrong due to the dynamics of the world. *Because there is always at least one linearization that will get us to the goal, the POP Method always succeeds.*

Critical Orderings

A *critical ordering* is any pair of unordered actions in a POP that must be ordered in a particular way for the Sequential Method to work well. If two linearizations differ only in the ordering of the critical pair of actions, then the Sequential Method for one

linearization will outperform the Sequential Method for the other. Because the POP Method simultaneously handles all linearizations, the ordering is irrelevant. We construct the *Tail* domain such that k sequential actions each provide a single precondition to the "tail" action. There is a "head" action, however, that follows the k initial actions and can produce all of the required preconditions for the tail action. The only two actions left unordered are the head and tail actions. An example with $k = 3$ is shown in Figure 4.9, and the full domain description can be found in Appendix B.3.

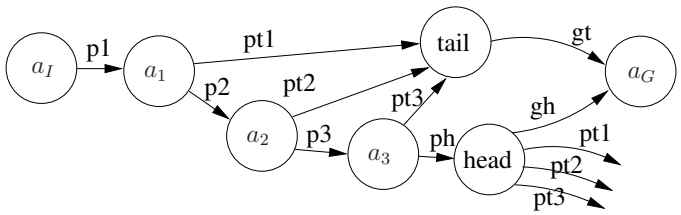


Figure 4.9: Example of the Tail domain with $k = 3$. An edge with no endpoint indicates an unused action effect.

Unlike the Dependent domain, it may be beneficial to have a given ordering even when the agent will always reach the goal. We investigate the performance of the two approaches in a simulation where at every time step, we set a randomly selected fluent to true or do nothing if it is already true. The POPs have the property that any linearization will reach the goal eventually in the simulation when using the Sequential Method. What is of interest is how *quickly* the agent can achieve the goal.

When using the Sequential Method for the linearization that has head ordered after tail, positive fluent changes have little impact on the number of steps required to reach the goal. The other linearization has the advantage of being able to *serendipitously jump* into a future part of the plan. We show the mean number of steps for each approach on eight instances in Figure 4.10.

We see a clear trend for the Sequential Method which suggests it requires roughly k actions to reach the goal. In contrast, the number of actions required on average for the POP Method grows very slowly – the final problem taking only a quarter of the actions to reach the goal on average. The POP Method is able to skip large portions of the plan when a fluent changes from false to true. The Sequential Method with the wrong linearization, on the other hand, must continue executing almost the entire plan.

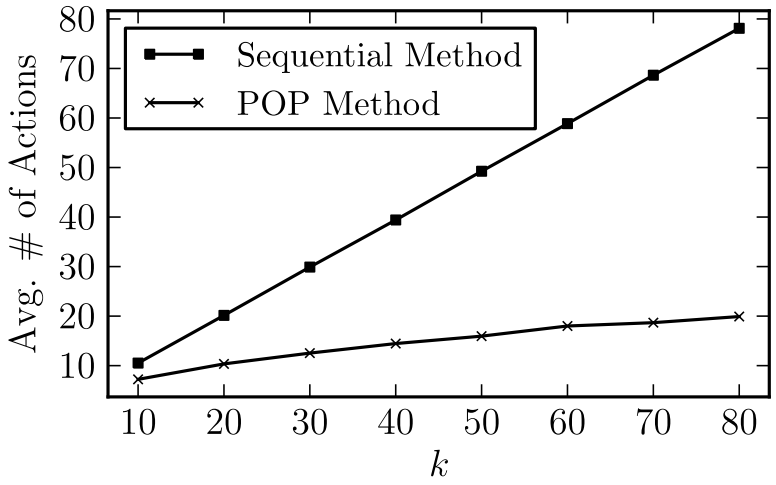


Figure 4.10: The mean number of actions executed before the goal was reached for 1000 trials on each problem in the Tail domain.

4.6.4 Analysis

The state coverage for the IPC domains was not as great as the coverage for the expository domains because the IPC domains impose significant constraints on action orderings and thus do not highlight the flexibility afforded by POPs and exploited in our EM approach. In general it is beneficial to use our approach, even for a single linearization. We have identified two scenarios, however, in which our approach fails. First, in the Parallel domain with k actions, there are $2^k - 1$ unique conditions. While this number is far smaller than the $k \cdot k!$ possible suffixes, it is large enough to become unwieldy for $k > 15$. Second, even if the POP is a single linearization, interactions between the ordering of the add effects and preconditions, along with the fluent ordering in the construction of the OADD, can result in an exponential increase in policy size. We have not, however, observed this behaviour in our experiments. Both scenarios suggest that in a richer domain, it would be interesting to investigate the trade-off between the size of the policy representation and its robustness.

4.7 Discussion

In this chapter we examined the problem of monitoring the execution of a POP. Due to its structure, a POP compactly represents a family of plans that share actions but allows for numerous different linearizations (exponentially many in the number of unordered pairs of actions). Our objective was to develop a means of POP EM that would seamlessly switch between these different plans based on the current state of the world, and thus

maximally exploit the flexibility of the POP. EM of sequential plans typically attempts to determine whether a plan remains valid with respect to the state of the world. We defined the objective of POP EM as determining POP *viability* and characterized the conditions under which a POP was viable by relating them to goal regression over all linearization suffixes; an approach that depends critically on the notion of state relevance.

Acknowledging the inefficiency of such a computation, we developed a more efficient algorithm that employed goal regression but exploited shared POP substructure to do so efficiently. We proved the correctness of this algorithm with respect to POP viability. Then, rather than employ these "relevant conditions" as input to a typical EM algorithm, we employed these conditions in the construction of a structured policy: an OADD that takes a state as input and returns an action to perform in reaction to the state of the world. In so doing, the policy combines two phases of EM – state evaluation and action selection – into a single unified system.

Here, we discuss some related work and conclude with potential future research.

4.7.1 Related Work

As noted in Section 4.1, there are a variety of approaches directly related to ours. Shakey the robot [34] and the more recent approach for sequential EM by Fritz and McIlraith [38] both characterize the conditions under which a partially executed sequential plan remains valid as the goal formula regressed through the plan that remains. The state of the world is compared to these conditions, and the smallest plan suffix is chosen for execution.

For a POP, SIPE [100] and Prodigy [96] are the most well-known systems to perform execution monitoring. One key difference between our method and theirs is that they monitor the violation of *causal links* and attempt to repair them as necessary. If a causal link becomes violated, these systems attempt to repair the plan or replan from what is known of the current state. The Prodigy system extended this approach to interleave planning and execution. Both SIPE and Prodigy monitor the validity of the entire POP that remains to be executed. In contrast, we would like to monitor the validity of any potential partial execution of the POP and not only a "current" partial execution. Doing so allows us to continue executing the POP from an arbitrary point in the plan.

One limitation of relying on the causal link structure of a POP is that causal links identify a single source of support for the existence of a necessary state condition. In cases where there is redundant support for a state condition, (i.e., where more than one actions realizes the necessary state condition) a violation of a causal link may not

necessitate a change in the plan because the necessary fluent will be achieved by another action. We avoid this problem with the regression-based approach.

One approach that takes the causal links a step farther is the EM in the Pike system [70]. The Pike system computes the causal links only as they are needed: if an un-executed action a requires p to hold, and the agent wishes to execute an action that deletes p, it will allow this under the condition that some other un-executed action in the plan adds p and can be ordered before a. While this approach overcomes some of the limitations that relying on causal links imposes, it is still less flexible than the execution scheme we propose in this chapter.

There are commonalities between our approach and work on the topic of dynamic controllability (e.g., Shah *et al.* [91]): our shared methodology is to compile plans into a dispatchable form for online execution. However, whereas the dynamic controllability work generally mitigates for uncertainty in the execution time for actions, our work addresses unexpected change in the environment. Our work complements dynamic controllability by focusing on a different source of uncertainty. In many real-world scenarios, both sources of uncertainty appear, and we intend to explore synergies between these two approaches. Chapter 5 presents the foundation for the potential merger of causal and temporal uncertainty.

Also related to our work are a number of systems for approximate probabilistic planning. For example, ReTrASE [60] uses regression in a similar fashion to build a "basis function" that provides the condition for which a plan succeeds in a deterministic version of a probabilistic planning problem. This use of determinization followed by regression is related to previous works by Sanner & Boutilier [90] and Gretton & Thiébaux [45] which use first-order regression on optimal plans over small problems to construct a policy for larger problems in the same domain. In a similar vein, Kaelbling and Lozano-Pérez use a form of regression to compute a plan that succeeds in a probabilistic setting with very large state spaces [52].

4.7.2 Conclusion

We evaluated our POP EM system both analytically and experimentally with the objective of assessing the claim that employing a POP, rather than a sequential plan, could provide notably enhanced flexibility at execution time. Experiments were run on IPC domains and on expository domains. On the IPC domains, which are designed to be more constrained than many real-world applications, our approach is able to continue executing in up to 2.5 times the number of states as the sequential-plan-based approach.

The speed-up in identifying an action of our POP Policy compared to the sequential-plan-based approach is up to a factor of 17, indicating a significant benefit for using the policy in a real-time plan execution setting. Finally, our expository domains highlight various properties that affect POP EM. In these domains, we demonstrated an exponential increase in the number of states where the POP remains viable relative to the sequential counterpart; demonstrating the robustness afforded by generalizing a POP.

There are a few extensions to our approach which we hope to pursue. When the POP Method is unable to return an action, we could instead attempt to re-plan using the information already embedded in the policy. Doing so opens the door to a variety of re-planning strategies, and suggests there may be merit in finding a more intelligent construction of the OADD. There is also the potential to develop a better informed heuristic for computing a sequential plan that will produce a POP with the properties that lead to good performance in our EM approach.

Chapter 5

Robustly Executing Temporally Constrained Plans

5.1 Introduction

In Chapter 4 we presented a method for robustly executing a POP. When an artificial agent is realized and must act in the real world, however, the agent should consider factors beyond just what is true or false in the environment. Often, actions and events have complex temporal requirements that must be adhered to. For example, the area of *schedule dispatching* addresses the task of deciding *when* a set of events must occur in order to satisfy a set of temporal constraints [28]. In this chapter, we bridge the gap between plan execution and schedule dispatching by presenting a unified representation of a plan with temporal constraints, along with a well-defined semantics for valid executions. Combining techniques from the planning and scheduling communities is a challenging but important endeavour [93], and using a unified formalism to address both what is causally relevant and what is temporally relevant for a plan to succeed is central to the task of executing plans in the real world.

To address the need for plans that involve temporal restrictions, Dechter *et al.* proposed a representation called a *Simple Temporal Network* (STN) [28]. An STN consists of a set of instantaneous events and a set of simple temporal constraints that restrict the timing between a pair of events in the STN. The STN formalism is the foundation for research in the field of dispatching (i.e., executing) temporal networks (e.g., Muscettola *et al.* [78] and Vidal [97]), and STNs are pervasive in the planning community for the planners that employ some form of temporal reasoning (e.g., Younes and Simmons [104] and Coles *et al.* [18]). Because of their rich history and practical appeal, we adopt many

of the techniques introduced for STNs as components for our approach.

From the perspective of executing a synthesized plan, there are two key drawbacks of the STN formalism: (1) STN dispatching does not include a notion of state, either in how the world changes or what must be true for an event to be executed, and (2) STN dispatching assumes that every event will be executed exactly once. Following the contributions of Chapter 4, we aim to resolve both drawbacks by combining the techniques for robustly executing a POP with a new temporal constraint formalism that addresses the execution of a plan directly. Similar to Chapter 4, the driving motivation of our techniques is to is avoid costly replanning by allowing temporally restricted actions to be executed multiple times or not at all.

We develop a means of robustly executing a flexible plan-and-schedule representation through a filtering algorithm that enables an agent to accommodate and capitalize on calamitous and serendipitous changes to the state of the world. Our approach allows the agent to propose actions to realize the planning goal, on time, in many cases where a traditional execution monitoring system would be forced to perform costly and time-consuming re-planning or re-scheduling. As input to our approach, we begin with a POP and a set of temporal constraints. In this work we only consider using a POP as our input plan, but the principles underlying our technique are equally applicable to other plan representations. We use temporal constraints to restrict our plan for schedule dispatching. To this end, we propose a temporal constraint language somewhat in the spirit of PDDL3.0 [40], and whose semantics augments and extends the temporal constraints representable in STNs to deal with the ambiguities introduced by the repeated execution of actions in a plan.

To produce a robust executor of a plan with temporal constraints, we build on the work from Chapter 4 to identify the subset of a state that is relevant to the continued validity of plan linearizations induced by our POP. We further develop the idea of relevance to include the aspects of an agent's execution trace that must be considered for temporal viability: a notion we refer to as *temporal relevance*. We employ the techniques of Muscettola *et al.* [78] to assess the viability of the temporally relevant subset of actions executed by the agent, along with the actions yet to be performed.

The algorithms we develop exploit state relevance to transform our POP and temporal constraints into a policy that is ready for dispatching. This policy generalizes the linearizations induced by the original POP in the same way that the methods from Chapter 4 do, and we augment the policy with a filtering algorithm to eliminate the linearizations that violate the plan's temporal constraints. We employ a variety of temporal reasoning methods for the filtering, allowing the agent to consider only a very restricted subset

of the execution history in order to determine if all of the temporal constraints will be satisfied after achieving the goal.

Typical execution monitoring systems execute actions in a plan in the order prescribed until the goal is reached or a discrepancy is detected. At this point, they trigger replanning or rescheduling [68, 20, 70]. The dispatching of STNs operates in a similar fashion [28]. Intuitively, if some fragment of the plan can still achieve the goal, then execution should continue uninterrupted. Our approach seamlessly elects to execute parts of a plan multiple times if required and/or omits actions that are no longer necessary for achieving the goal. Such flexibility can introduce ambiguity in the interpretation of temporal constraints. For example, if the user must start to eat between 3 and 10 minutes after heating his/her dinner and eating is delayed causing him/her to re-heat, then what temporal relationship(s), if any, should exist between the first heating and the eating?

The set of temporal constraints we introduce resolves these execution-time ambiguities by being described in terms of the execution trace of a plan rather than over the actions in a plan. A further contribution of our temporal constraint formalism is a connection between the execution of actions and fluents in the state of the world. Having a temporal constraint between the execution of an action and conditions over state provides a compelling avenue for many of the temporal constraints we might want to express.

We formally define the syntax and semantics of the temporal constraints and prove the correctness of our methods with respect to the correct interpretation of our original POP and temporal constraints. We also empirically compare our approach with STN dispatching and a restricted version of our method. Compared to conventional execution monitoring systems, our approach has the potential to avoid replanning exponentially more often (in the size of the state). Our results serve to illustrate that we retain the flexibility of our previous work while adhering to the temporal constraints up to an order of magnitude faster than a naïve approach that does not compile the relevant temporal conditions in advance. Experiments with a simulated uncertain environment show that our approach achieves the goal in 92% of the trials while the standard STN dispatching technique has a success rate of roughly 30%.

Our contributions leverage existing work from both partial-order plan execution and temporal reasoning. While many of the core algorithms are based on existing techniques, our main contribution stems from the dynamic creation of temporal subproblems that need to be solved during execution. Our approach is noteworthy for its novelty and broad applicability while taking an important step towards integrating plan execution and schedule dispatching – tasks that are traditionally addressed independently [93]. Bringing the notions of plan execution and schedule dispatching together is central to

leveraging the advances made in their respective research areas.

5.1.1 Overall Approach

We propose an execution module, TpopExec, which is comprised of two components: 1) Compiler, an offline preprocessor that takes as input a POP and a set of temporal constraints, and produces a compiled representation; and 2) Executor, an online component that soundly selects a temporally consistent, *valid* plan fragment from the compiled plan that can still achieve the goal, if such a fragment exists. TpopExec does no replanning or repair. Rather, it can serve as a component of a larger execution engine to reduce, but not eliminate, the need for replanning.

TpopExec reacts to the state of the world, proposing the next action of one of a large number of valid plan fragments whose starting state satisfies the (small number of) necessary conditions for plan validity. The fragment chosen is the best quality one among all those that can achieve the goal while satisfying every temporal constraint. Such flexibility in plan execution and schedule dispatching is not found in existing methods without explicit replanning or rescheduling.

5.1.2 Contributions

The main contributions of this chapter are as follows:

- We present a novel formalism that incorporates plan execution and schedule dispatching. To address the possibility of a plan fragment being executed more than once, we introduce a variety of temporal constraints and provide a formal semantics for when an execution trace satisfies a temporal constraint. The constraints we introduce can be defined between a pair of actions, as well as between an action and fluent.

- We show how many of the more complex constraints can be reformulated as simpler constraints. This simplifies the exposition of our techniques, and demonstrates how we might reformulate future additions to the language in a similar fashion.

- By focusing on what is temporally relevant for both a plan fragment and an existing execution trace to satisfy every temporal constraint, we propose a method for compiling a temporally augmented POP. If a linearization of some fragment of the plan can achieve the goal while satisfying every temporal constraint, given the execution so far, then the compiled plan will use such a linearization. We achieve the compilation through the novel use of a number of schedule dispatching techniques.

- We present an online execution strategy that makes use of our compiled representation. We first retrieve every plan fragment that can achieve the goal for at least one arbitrary execution trace, and then find the best quality fragment that is also temporally viable with the actual trace (i.e., every temporal constraint is, or can be, satisfied).

- We prove several key properties about the execution framework we introduce. Most importantly, we show that our method of execution produces the expected behaviour of the system – continuing to execute in a correct manner whenever the goal is still achievable. We also show that the temporally relevant subset of the execution trace suffices for correctness of our approach.

- We present a prototype system to demonstrate the feasibility of our techniques. The system, TPOPEXEC, demonstrates the effectiveness of our temporal reasoning and the improvement in execution robustness afforded by our techniques. TPOPEXEC also employs a number of preprocessing techniques to help avoid expensive reasoning online.

The work in this chapter is based on a publication at IJCAI'13 [73]. This chapter contains the introduced algorithms in greater detail, includes an additional temporal constraint (cf. the **sometime-before** constraint in Section 5.3.2), and expands on the evaluation of the implemented system to assess the impact of preprocessing.

5.1.3 Organization

We begin this chapter with a review of schedule dispatching notation and a motivating example. In Section 5.3 we formally introduce the set of temporal constraints we allow and how they are interpreted over the execution of a plan. In Sections 5.4 and 5.5 we describe our method for compiling a plan with temporal constraints and our method for executing the compiled plan respectively. Section 5.6 describes the formal notion of temporal relevance that we leverage. We present an evaluation of our system in Section 5.7 and conclude with a discussion of related work and summary in Section 5.8.

5.2 Preliminaries

In this section, we present the background formalism and notation required for the temporal reasoning aspects of our approach. We cover Simple Temporal Networks (STNs) and present the standard algorithm for dispatching a temporal network that has already

been preprocessed for online execution. We also present a motivating example domain for the application of our techniques.

STNs are used extensively in the scheduling community, but for our purposes they have a few key limitations: (1) STNs do not consider state, either in the constraints defined or for checking the execution validity, and (2) STNs are not sufficiently expressive to describe constraints over the execution of a plan when an action or event appears multiple times. Despite the limitations of using STNs naïvely, we will make heavy use of them as a component of our approach.

5.2.1 Simple Temporal Networks

A Simple Temporal Network (STN) consists of a set of instantaneous events, and a set of simple temporal constraints over those events [28]. Typically, every event in an STN must be executed exactly once, but we will relax this assumption for the dispatching phase of our work. We use X_* to signify an event, and make the distinction between an action a and an event X_a corresponding to an execution of a. To simplify the notation in this section, we will use numeric indices for events (e.g., $X_1, X_2, \cdots$). We use the function $t(X)$ to indicate the time event X is executed. If defined for every event, the function $t(\cdot)$ is called a *schedule*. A *simple temporal constraint* restricts the timing of two events relative to one another:

Definition 10 (Simple Temporal Constraint)**.** For events X_1 and X_2, $[l, u]_{X_1,X_2}$ is the simple temporal constraint that restricts the timing of events X_1 and X_2 to be,

$$l \leq t(X_2) - t(X_1) \leq u$$

The values of l and u may be any real number (possibly negative, and including $\pm\infty$), but must satisfy $l \leq u$. We will use ϵ to designate an arbitrarily small amount of time, and assume that if $u - l < \epsilon$, then $u = l$.

Note that the simple temporal constraint does *not* give an indication of the absolute time that an event must occur. Rather, it simply indicates the relative timing between two events. The constraint $[l, u]_{X_1,X_2}$ can be interpreted as "event X_2 must occur no earlier than l time units and no later than u time units after X_1 occurs". We now have the notation required to formally define an STN:

Definition 11 (Simple Temporal Network [28])**.** A *Simple Temporal Network* (STN), $\langle E, C\rangle$, is a tuple that contains a set of events E and set of simple temporal constraints

C defined over E. An STN $\langle E, C\rangle$ is *consistent* iff there exists a schedule $t(\cdot)$ of the events E that satisfies every constraint in C.

Example 1 (STN)**.** Figure 5.1 shows an example of an STN with four events $E = \{X_1, X_2, X_3, X_4\}$, and four simple temporal constraints amongst the events:

$$C = \{[0, 10]_{X_1,X_2}, [0, 10]_{X_1,X_3}, [1, 1]_{X_2,X_4}, [2, 2]_{X_3,X_4}\}$$

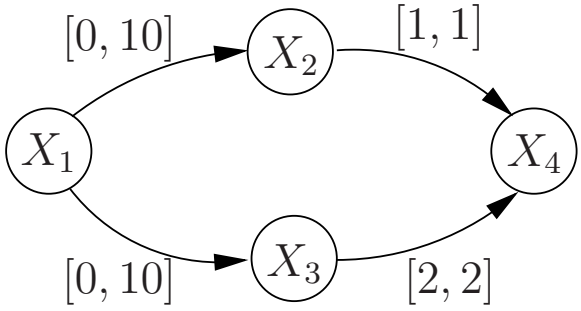


Figure 5.1: Example STN.

We typically include a starting event for every network to indicate the "start of time" (X_1 in Figure 5.1). The starting event is assumed to be scheduled at time $t(X_1) = 0$.

One potential (naïve) approach for determining the consistency of an STN is to find a full schedule such that all of the constraints are satisfied. Returning to our example, we can see the STN is consistent by using the following setting for the timing of the 4 events: $t(X_1) = 0,\ t(X_2) = 4,\ t(X_3) = 3,$ and $t(X_4) = 5$. There are, however, sufficient and necessary conditions that are easier to compute for ensuring consistency.

The prevailing method for determining consistency of an STN is to run an all-pairs-shortest-path algorithm on a graph induced by the STN [28]: nodes correspond to events, and every temporal constraint $[l, u]_{X_1,X_2}$ induces the edges (X_1, X_2) and (X_2, X_1) with weights of $w((X_1, X_2)) = u$ and $w((X_2, X_1)) = -l$. We refer to the resulting shortest-path graph as the *distance graph*. If the distance graph contains a negative cycle, then the STN is inconsistent: some event must occur before itself. We will further make use of the distance graph, as its structure will be useful for dispatching the network and computing further bounds.

5.2.2 Schedule Dispatching

Given a consistent STN, *schedule dispatching* is the process of assigning time points to the STN's events in order of execution time. In a sense, schedule dispatching is the analogue

to plan execution. One useful advantage of schedule dispatching is that it provides a lower and upper bound on the execution time of the next event to be executed. We will use this later for the effective execution of our plans. In this section, we describe the standard approach for dispatching a consistent STN.

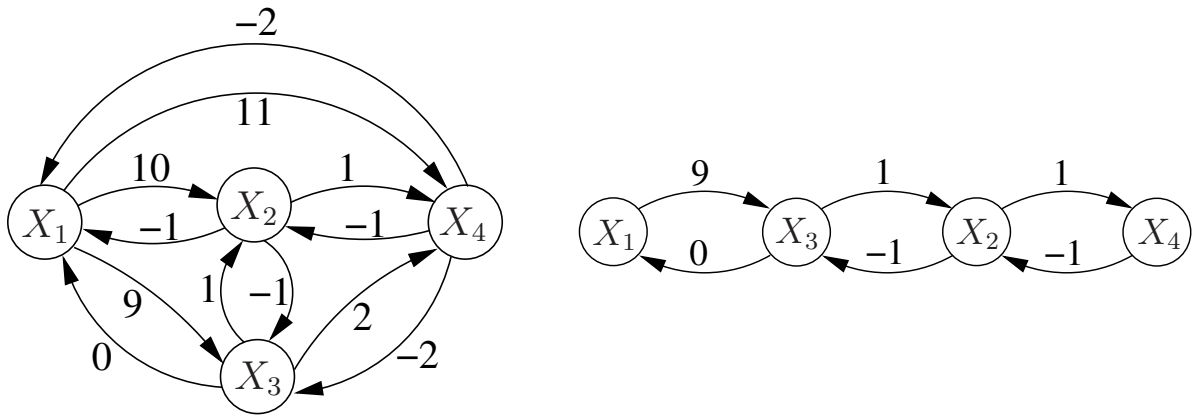
The typical method for checking STN consistency produces a distance graph in the process. If the STN is consistent, we can use the distance graph to dispatch the events in the STN while satisfying every constraint. An edge (X_1, X_2) with weight k in the distance graph represents the temporal requirement that $X_2 - X_1 \leq k$ [78]. The dispatching procedure that uses the distance graph is presented in Algorithm 8. Note that the complexity of propagation in the algorithm depends on the number of neighbours associated with an event.

Algorithm 8: STN Dispatching Algorithm [78]

Input: Distance graph, $\langle V, E, w \rangle$, where V is the set of events, E is the set of edges, and $w : E \to \mathbb{R}$ is a weighting of the edges such that $(X_1, X_2) \in E$ represents the constraint $X_2 - X_1 \leq w(X_1, X_2)$

1 $A = \{\text{start\_time\_point}\}, \quad t = 0, \quad \text{and } S = \emptyset$
2 $l(\text{start\_time\_point}) = u(\text{start\_time\_point}) = 0$
3 $\forall X \in V \setminus \{\text{start\_time\_point}\}, l(X) = 0 \text{ and } u(X) = \infty$
4 **while** $S \neq V$ **do**
5 Pick some $X_1 \in A$ such that $l(X_1) \leq t \leq u(X_1)$
6 $t(X_1) = t$
7 $S = S \cup \{X_1\}$
8 **foreach** $(X_1, X_2) \in E$ *and* $X_2 \notin S$ **do**
9 $u(X_2) = \min\{u(X_2), t + w(X_1, X_2)\}$
10 **foreach** $(X_2, X_1) \in E$ *and* $X_2 \notin S$ **do**
11 $l(X_2) = \max\{t, l(X_2), t - w(X_2, X_1)\}$
12 $A = \{X \in V - S \mid \forall (X, X') \in E, [w(X, X') < 0 \Rightarrow X' \in S]\}$
13 Wait until $\min_{X \in A} l(X) \leq t \leq \min_{X' \in A} u(X')$

In Algorithm 8, A contains the *activated* events (those events that are eligible to execute when the time fits their current bounds), S contains those events that have already been executed, and $l(X)$ (respectively $u(X)$) gives the lower (respectively upper) bounds on the time event X can be executed. Lines 5-7 pick an active event arbitrarily and execute it, while lines 8-11 propagate the bounds information to the direct neighbours of the event being executed. Line 12 sets A to be all those events that can now be executed (all events necessarily preceding them have been executed). Finally, line 13 waits for an



(a) Example of the all-pairs-shortest-path for a simple STN.

(b) Example of the distance graph for a simple STN.

Figure 5.2

unspecified but restricted length of time until we can at least execute some event, and no later than the minimum deadline of another event. This pause of arbitrary length gives the agent flexibility at execution time to delay events and possibly handle external contingencies.

As noted above, the efficiency of the dispatching algorithm is tied to the number of neighbours an event has (i.e., propagating the bounds). To improve on the dispatching complexity, we can process the distance graph by removing redundant edges. Using the observation that a distance graph must satisfy the triangle inequality, the following set of dominance relations allows us to soundly remove edges from the distance graph before dispatching [78]:

1. A non-negative edge (X_1, X_3) is *upper dominated* by another non-negative edge (X_2, X_3) iff $w((X_1, X_2)) + w((X_2, X_3)) = w((X_1, X_3))$

2. A negative edge (X_1, X_3) is *lower dominated* by another negative edge (X_1, X_2) iff $w((X_1, X_2)) + w((X_2, X_3)) = w((X_1, X_3))$

Any edge that is dominated (either upper or lower) is redundant and can be safely removed from the distance graph. Reducing the number of edges allows for quicker dispatching, as the number of loops on lines 8 and 10 in Algorithm 8 are reduced. To illustrate the potential of reductions, consider the STN in Figure 5.1. The all-pairs-shortest-path algorithm results in the distance graph in Figure 5.2a. After removing the redundant edges, we are left with the distance graph shown in Figure 5.2b.

In our work we assume that the final distance graph we work with is in a minimal form that contains no redundant edges.

5.2.3 Motivating Example

We want to build a system capable of executing a temporally constrained plan in a dynamic environment. Existing execution monitoring systems (e.g., our work from Chapter 4) allow us to robustly execute a plan, but do not facilitate the specification of temporal constraints. Conversely, schedule dispatching techniques allow us to execute a temporally restricted plan, but do not allow for reasoning during execution about why an action is required. The aim of our work is to enable the robust execution of a temporally constrained plan. Here, we present a scenario that motivates such an executor.

Consider the TpopExec for a mobile-phone based Cognitive Assistant (CA) that oversees a user's daily activities. The CA knows the user's plan for the day including activities such as laundry, transportation, dinner, and a movie. It is updated about the state of the world by the user, RSS feeds, etc. CA "executes" a plan by reminding the user to perform actions. As the state is updated, CA's Executor revises its reminders. If the user's date is delayed, he/she may need to re-book dinner. If the user's friend gives them $50, he/she will not need to get cash from the bank.

As noted in Section 5.1, TpopExec is able to seamlessly re-execute or omit portions of a plan. This re-execution or elimination of actions can create ambiguity in the interpretation of standard STNs, as illustrated by the "heating and eating" example. It is also the case that many temporal constraints are expressed more compellingly as constraints between state properties and actions (e.g., "Be at the movie at least 15 minutes before it starts.") rather than solely between actions. These two desiderata serve as motivation for our new language. We have created the CA domain for testing our implementation, and further details on the actions and constraints included in the domain can be found in Appendix B.4.

5.3 Temporally Constrained POP (TPOP)

The starting point for our work is a *Temporally Constrained POP* (TPOP), $\langle\langle \mathcal{A}, \mathcal{O}\rangle, \mathcal{C}\rangle$, which comprises a valid POP $\langle \mathcal{A}, \mathcal{O}\rangle$, and a set of temporal constraints $\mathcal{C}$ (defined further in Section 5.3.2). Here, we do not concern ourselves with where the POP comes from, but options include using a partial-order planner (e.g., VHPOP [104]), relaxing a sequential plan (e.g., the work presented in Chapter 3), or having a POP specified by the user.

In this work, we assume that the temporal constraints originate with the user, with the exception of those generated in the transformation of durative actions to pairs of instantaneous actions [35]. This decoupling enables a POP to be re-used in multiple

scenarios by simply varying the temporal constraints. For example, a common plan can be used for multiple agents, each with a set of agent-specific temporal constraints.

In the remainder of this section, we introduce the execution framework that we will consider and a set of temporal constraints that addresses the ambiguity inherent in a framework as expressive as ours. We end with a discussion on how we can compile away a subset of the temporal constraints to simplify the exposition of our techniques.

5.3.1 Execution Traces

We will define the semantics of our temporal constraints with respect to the execution trace of the plan – a history of action-state pairs that are indexed by time and represented as a *timed word* [2].

Definition 12 (Trace)**.** Given a TPOP $\langle\langle \mathcal{A}, \mathcal{O}\rangle, \mathcal{C}\rangle$ and planning problem $\langle F, O, I, G\rangle$, we define a *trace*, $\mathcal{T}$, to be a finite timed word, $(\sigma_0, t_0), \cdots, (\sigma_n, t_n)$, where $\sigma_i \in \Sigma$, the alphabet Σ ranges over $\mathcal{A} \times \mathcal{S}$, and the time values, $t_i \in \mathbb{R}_{\geq 0}$, are strictly increasing. $\mathcal{T}$ is *executable* iff for every $((a_i, s_i), t_i)$ in $\mathcal{T}$, a_i is executable in s_i. $\mathcal{T}$ is *static* iff for every $((a_i, s_i), t_i)$ in $\mathcal{T}$, if $i > 0$ then s_i is the result of executing a_{i-1} in state s_{i-1}. We signify the concatenation of traces $\mathcal{T}$ and $\mathcal{T}'$ as $\mathcal{T} \cdot \mathcal{T}'$.

We assume that the state of the world is fully observable. If a fluent changes unexpectedly (e.g., through an exogenous event), a tuple in the trace reflects this change. A common technique generally employed for this situation is to assume that there is a sensing action in the trace to reflect the fact that the fluent changes [59]. Alternatively, we can assume that a sensing action occurs at a regular frequency.

A single action $a \in \mathcal{A}$ may appear more than once (i.e., we may have $a_i = a_j$ and $i \neq j$), and unless the trace is static, the states need not correspond to the progression of previous action-state pairs (i.e., s_{i+1} is not necessarily equal to $\mathcal{P}(s_i, a_i)$ – the expected state after executing a_i in state s_i). A single tuple $((a_i, s_i), t_i) \in \mathcal{T}$ is an *occurrence*, and we refer to the actual trace of performed actions as an *execution trace*.

The restriction that we require no two events to share the same time point is not overly prohibitive, as a number of actions can occur within ϵ time of one another. In the next section we formally describe the temporal constraints, but for now we say that a constraint can be *satisfied* with respect to a trace. An execution trace is *valid* if it is executable, satisfies every temporal constraint, and the final action is a_G (i.e., the goal is achieved). Finally, we say that an execution trace is a *valid partial trace* if it can be extended with zero or more events to be a valid trace.

5.3.2 Temporal Constraints

For robust execution we may want to execute an action multiple times. For example, if the intended effect of an action is undone by some exogenous event or change in the world, then the action should be executed again. The user does not know in advance which actions TpopExec will execute, and so specifying temporal constraints directly on the action occurrences in the execution trace is not meaningful. Instead, we allow constraints of various types to be specified on actions in the POP that are interpreted over an execution trace. The semantic interpretation of our constraints serve to disambiguate how a pair of action occurrences in the execution trace should be temporally restricted.

The constraints we allow fall into two categories: action-centric and state-centric. The action-centric constraints provide a compelling range of constraints that generalize those found in the scheduling literature, while the state-centric constraints extend the traditional notion of temporal constraints to involve the state of the world. To avoid confusion, we will use an indexed action to signify it is part of an *occurrence* (e.g., a_i) and a, b for actions in the input TPOP.

Inspired by PDDL3.0 [40] and linear temporal logic [83], our language introduces five temporal modal operators ranging over the actions of our TPOP, the fluents in our planning domain, and time (the positive reals). Our language supports the specification of a set of constraints, but no connectives. During execution, actions in our TPOP may be repeated or skipped to accommodate unexpected changes in the environment, requiring a formalism strictly more expressive than STNs: in an STN, there is no accommodation for unplanned re-execution or omission of actions, nor is there a facility to express temporal constraints with respect to the state of the world [28]. We define the five temporal constraints informally as follows:

Definition 13 (Temporal Constraint Types)**.**

- A *past constraint* between actions a and b over bounds $l, u \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ stipulates that if b occurs, then a must have occurred previously and the most recent occurrence of a is between l and u time units in the past:

  (latest-before b a l u**)**

- An *existential past constraint* between actions a and b over bounds $l, u \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ stipulates that if b occurs, then a must have occurred previously and one such

occurrence of a is between l and u time units in the past:

(sometime-before b a l u**)**

- A *future constraint* between actions a and b over bounds $l, u \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ stipulates that if a occurs, then b must occur in the future and the next occurrence is between l and u time units in the future:

(earliest-after a b l u**)**

- A *past fluent constraint* between an action a and fluent f over bounds $l, u \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ stipulates that if a occurs, then f must have held between l and u time units in the past:

(holds-before a f l u**)**

- A *future fluent constraint* between an action a and fluent f over bounds $l, u \in \mathbb{R}_{\geq 0} \cup \{\infty\}$ stipulates that if a occurs, then f must hold between l and u time units in the future:

(holds-after a f l u**)**

The notation mirrors the PDDL3.0 preference syntax: e.g., **(latest-before** b a l u**)** should be read as "the latest occurrence of action a before an occurrence of b is between l and u time units". Consider the temporal constraint that restricts the time between heating and eating a meal, **(latest-before** *eat_meal heat_meal* 3 10**)**: one should wait at least 3 minutes for a meal to cool, but it should be heated within 10 minutes prior to eating. One can heat the meal multiple times prior to eating, but only the most recent heating will be considered for the temporal constraint. Additionally, if after eating some of the food, the user decides to eat some more food at a later time (i.e., they unexpectedly become hungry), the timing of the second eating action must be consistent with the constraint and the timing of the most recent heating action. If the constraint had been **(earliest-after** *heat_meal eat_meal* 3 10**)**, there would be no constraint between heating and the second occurrence of eating: that constraint is only relevant to the *first* occurrence of eating after a meal is heated.

Similarly, consider the action of starting to watch a movie and the fluent correspond-

ing to being at the mall. The constraint **(holds-before** *watch_movie at_mall* 10 ∞**)** stipulates that one must be at the mall at least 10 minutes before the movie begins. If we have an action for waking up and a fact corresponding to being full, then the constraint **(holds-after** *wake_up full* 0 60**)** stipulates that within one hour of waking up, one must be full. This could be because enough was eaten the night before, or breakfast was eaten.

We refrain from including a **sometime-after** constraint, because it does not offer any further expressiveness beyond the **earliest-after** constraint in our formalism. The target behaviour of TpopExec is to find a partial plan that can achieve the goal while satisfying every temporal constraint. With such an approach, we will only succeed under the assumption that the actions in the partial plan will be executed exactly once (even if this is not the case during the actual execution). Because of the choice in target behaviour, TpopExec cannot treat the **sometime-after** constraint any different from the **earliest-after** constraint.

For both variants of a future constraint, an execution trace may be a valid partial trace but not a valid trace because the constraint is not yet satisfied. We refer to such constraints as being *unresolved*.

Note the absence of a temporal constraint between fluents. Such constraints are not included because the interpretation of the temporal constraints requires some element to have a universal scope: e.g., "whenever a occurs then $\cdots$". Our interpretation of temporal fluent constraints is not amenable to describing constraints such as "whenever f holds then $\cdots$" because we model the observation of a fluent as an instantaneous event. Notice that we further do not allow **(latest-before** f a l u**)** and **(earliest-after** f a l u**)** for the same reason.

If $a_\vdash$ and $a_\dashv$ denote the instantaneous actions corresponding to the start and end of a durative action a respectively, we augment the domain with **(latest-before** $a_\dashv$ $a_\vdash$ l u**)** where l and u are lower and upper bounds on the duration of a, to enforce the correctness of the translation. If a begins, it must be completed, and we cannot complete a unless it has begun: these properties are handled through the standard encoding used to split a durative action. Following Fox and Long [35], any durative action appears in the planning domain as a pair of instantaneous *start* and *end* actions that must alternate. Currently, we are able to handle domains where durative actions must overlap, typically referred to as *required concurrency* [36, 22], but we do not handle situations where a single durative action must overlap with itself during execution [19]: when a *start* action occurs, the corresponding *end* action must occur before the *start* can be executed again. Our work is focused on STRIPS planning problems with durative actions compiled away and a set

of temporal constraints inspired by those found in PDDL3.0. STRIPS cannot require that two instantaneous actions "execute in parallel" (e.g., Boutilier and Brafman [13]), so this situation will not arise. In the future, we hope to expand to other aspects of PDDL including conditional effects, numeric state fluents, and continuous change.

Figure 5.3 defines the semantics of our temporal modal operators with respect to a trace. Notationally, we use i, j, k to represent indices in the trace and a_*, t_*, s_* to represent the actions, time points, and states of a particular *occurrence* respectively (for $* \in \{i, j, k\}$). To simplify formulae common to many constraints, we assume that $t_i = -\epsilon$ for all $i < 0$ and use the following abbreviations:

- Action a occurs at index i in the trace:

$$(\textbf{occ}\ a\ i) \stackrel{\text{def}}{=} a_i = a$$

- The time between indices i and j is bounded between l and u:

$$(\textbf{time-diff}\ i\ j\ l\ u) \stackrel{\text{def}}{=} l \leq t_j - t_i \leq u$$

5.3.3 Action-centric Temporally Constrained POP (ATPOP)

To simplify the exposition, we present our approach for a subclass of TPOPs where the constraints, $\mathcal{C}$, are restricted to involve only the **latest-before** and **earliest-after** constraints: referred to as an action-centric temporally constrained POP (ATPOP). We assume that the ATPOP is provided to TPOPEXEC. Potential sources of an ATPOP include manually hand-coding one, annotating a standard POP with temporal constraints, or computing one with a dedicated planner. For this work, however, we focus on the task of executing an ATPOP, rather than synthesizing one.

The approaches in Sections 5.4 and 5.5 apply only to ATPOPs. A feature of our approach is that we are able to leverage the duality between actions and fluents to reformulate TPOPs involving state-centric constraints into ATPOPs. Doing so enables the elegant application of our approach to arbitrary TPOPs. The reformulation is sound, but incomplete with respect to the **sometime-before** and **holds-before** constraints. For completeness, we would need to expand the temporal reasoning to handle disjunctive constraints to address the **sometime-before** constraint. We could subsequently compile a **holds-before** constraint into a **sometime-before** constraint so that the reasoning is performed entirely on action execution. Here, we describe how the remaining constraints

$$((a_0, s_0), t_0), \cdots, ((a_n, s_n), t_n) \models \textbf{(latest-before } b\ a\ l\ u\textbf{)}$$
$$\text{iff } \forall j : 0 \leq j \leq n \text{ if } \textbf{(occ } b\ j\textbf{)} \text{ then}$$
$$\exists i : 0 \leq i < j,\ \textbf{(occ } a\ i\textbf{)} \wedge \textbf{(time-diff } i\ j\ l\ u\textbf{)} \wedge \forall k : i < k < j,\ a_k \neq a$$

$$((a_0, s_0), t_0), \cdots, ((a_n, s_n), t_n) \models \textbf{(sometime-before } b\ a\ l\ u\textbf{)}$$
$$\text{iff } \forall j : 0 \leq j \leq n \text{ if } \textbf{(occ } b\ j\textbf{)} \text{ then}$$
$$\exists i : 0 \leq i < j,\ \textbf{(occ } a\ i\textbf{)} \wedge \textbf{(time-diff } i\ j\ l\ u\textbf{)}$$

$$((a_0, s_0), t_0), \cdots, ((a_n, s_n), t_n) \models \textbf{(earliest-after } a\ b\ l\ u\textbf{)}$$
$$\text{iff } \forall i : 0 \leq i \leq n \text{ if } \textbf{(occ } a\ i\textbf{)} \text{ then}$$
$$\exists j : i < j \leq n,\ \textbf{(occ } b\ j\textbf{)} \wedge \textbf{(time-diff } i\ j\ l\ u\textbf{)} \wedge \forall k : i < k < j,\ a_k \neq b$$

$$((a_0, s_0), t_0), \cdots, ((a_n, s_n), t_n) \models \textbf{(holds-before } a\ f\ l\ u\textbf{)}$$
$$\text{iff } \forall j : 0 \leq j \leq n \quad \text{if } \textbf{(occ } a\ j\textbf{)} \text{ then}$$
$$\exists i : 0 \leq i \leq j,\ (f \in s_i) \wedge [\exists t^* : (t_{i-1} < t^* \leq t_i) \wedge (l \leq t_j - t^* \leq u)]$$

$$((a_0, s_0), t_0), \cdots, ((a_n, s_n), t_n) \models \textbf{(holds-after } a\ f\ l\ u\textbf{)}$$
$$\text{iff } \forall i : 0 \leq i \leq n \text{ if } \textbf{(occ } a\ i\textbf{)} \text{ then}$$
$$\exists j : i < j \leq n,\ (f \in s_j) \wedge [\exists t^* : (t_{j-1} < t^* \leq t_j) \wedge (l \leq t^* - t_i \leq u)]$$

Figure 5.3: Semantics of the temporal modal operators with respect to a trace. $l, u \in \mathbb{R}_{\geq 0} \cup \{\infty\}$, $l \leq u$, and a, b are actions.

can be compiled away.

The first step of the reformulation is to treat every **sometime-before** constraint as a **latest-before** constraint. This effectively prohibits having an action that will repeat in multiple past temporal constraints – in an ATPOP the various constraints must all synchronize on the most recent occurrence of the leading action.

Next, we reformulate our state-centric constraints to action-centric constraints. For each fluent participating in a temporal constraint, we introduce an auxiliary action a^f with the precondition of $PRE(a^f) = \{f\}$ and no add or delete effects. These actions are used to record the observation of a fluent. We then replace the state-centric constraints with suitable action-centric counterparts: **(holds-before** $a\ f\ l\ u$**)** (respectively **(holds-after** $a\ f\ l\ u$**)**) is replaced by the constraint **(latest-before** $a\ a^f\ l\ u$**)** (respec-

tively (**earliest-after** a a^f l u)).

This modification permits TpopExec to observe necessary facts when it needs to. Finally, we require a unique auxiliary action for each state-centric constraint, as sharing the auxiliary actions is unsound: using the same action could cause a temporal inconsistency.

5.4 Compiling ATPOPs

TpopExec is composed of two main components: Compiler and Executor. In this section, we describe the suite of techniques that make up the Compiler component.

5.4.1 General Approach

Given an ATPOP, execution trace, and state of the world, TpopExec needs to determine if any fragment of the ATPOP can achieve the goal and satisfy all of the temporal constraints while taking the execution trace so far into account. We compile the causal and temporal conditions required for every partial plan into a policy that indicates whether or not we can still reach the goal, and if so, which action to execute next and when. The key component of the policy representation is a *partial plan context*.

Definition 14 (Partial Plan Context)**.** Given a planning problem $\langle F, O, I, G\rangle$ and an ATPOP $\langle\langle \mathcal{A}, \mathcal{O}\rangle, \mathcal{C}\rangle$, we define a *partial plan context* as any tuple $\langle \mathcal{A}_\dashv, \mathcal{O}_\dashv, \psi, a\rangle$, where:

1. $\mathcal{A}_\dashv \subseteq \mathcal{A}$ is the set of actions to be executed.
2. $\mathcal{O}_\dashv$ is a set of ordering constraints over $\mathcal{A}_\dashv$.
3. $\psi \subseteq F$ is a set of fluents sufficient for executing $\mathcal{A}_\dashv$.
4. $a \in \mathcal{A}_\dashv$ and $\nexists a' \in \mathcal{A}_\dashv$ *s.t.* $(a' \prec a) \in \mathcal{O}_\dashv$

We refer to the *corresponding suffix* of a context to be the POP suffix of $\langle \mathcal{A}, \mathcal{O}\rangle$ that contains the actions $\mathcal{A}_\dashv$. *Context Viability* captures the notion that there exists a linearization of the partial plan context's POP that is valid and satisfies every constraint.

Definition 15 (Context Viability)**.** Given a planning problem Π, ATPOP $\langle\langle \mathcal{A}, \mathcal{O}\rangle, \mathcal{C}\rangle$, valid partial trace $\mathcal{T}$, and current state of the world s, a partial plan context $\langle \mathcal{A}_\dashv, \mathcal{O}_\dashv, \psi, a\rangle$ is (1) *causally viable* with respect to Π and s iff the POP $\langle \mathcal{A}_\dashv, \mathcal{O}_\dashv\rangle$ has a linearization starting with a that is a valid plan for the planning problem with s as the initial state, (2) *temporally viable* with respect to $\mathcal{C}$ and $\mathcal{T}$ iff there exists a trace $\mathcal{T}'$ where the actions in

$\mathcal{T}'$ correspond to a linearization of $\langle \mathcal{A}_{\dashv}, \mathcal{O}_{\dashv} \rangle$ and $\mathcal{T} \cdot \mathcal{T}'$ satisfies every temporal constraint in $\mathcal{C}$, and (3) simply *viable* with respect to Π, s, $\mathcal{C}$, and $\mathcal{T}$ iff $\langle \mathcal{A}_{\dashv}, \mathcal{O}_{\dashv} \rangle$ has a linearization that allows the context to be both causally and temporally viable.

The job of Compiler is to compute every partial plan context and determine the sufficient conditions for each one to be a viable context. This computation is done offline, prior to execution, and therefore both the current state of the world and execution trace are unknown. Compiler stores additional information for each context to make the job of Executor easier when the state of the world and execution trace becomes available.

5.4.2 Causal Viability

Searching for a viable context at runtime is computationally infeasible. Instead, to generate every causally viable context, we appeal to the approach from Chapter 4 which transforms a POP into a policy by way of generalizing the input plan. Algorithm 6 in Section 4.4 produces a sequence of condition-action pairs where the condition holds in a state iff some linearization of the POP has a suffix that can reach the goal starting with the action. This condition is precisely what we need for determining the causal viability of a context. However, rather than augment the condition-action pairs directly, we must modify the algorithm to generate partial plan contexts so that the temporal reasoning is sufficiently informed of the ordering constraints that are necessary for a valid execution.

We modify Algorithm 6 to construct partial plan contexts. ContextGenerator, our modified approach, is shown in Algorithm 9. ContextGenerator additionally uses the following notation:

- MakeContext$(\langle \mathcal{A}, \mathcal{O} \rangle, \mathcal{A}_{\vdash}, \mathcal{O}_*, a, \psi, N)$: A procedure that returns a carefully constructed POP, $\langle \mathcal{A}_{\dashv}, \mathcal{O}_{\dashv} \rangle$. $\mathcal{A}_{\dashv}$ includes a and all of the actions not found in $\mathcal{A}_{\vdash}$. $\mathcal{O}_{\dashv}$ includes (1) all of the ordering constraints in $\mathcal{O}_*$, (2) all of the ordering constraints originating from a, and (3) all of the ordering constraints that correspond to a supporting a fluent still needed in N:

$$\begin{aligned} \mathcal{A}_{\dashv} &= (\mathcal{A} - \mathcal{A}_{\vdash}) \cup \{a\} \\ \mathcal{O}_{\dashv} &= \mathcal{O}_* \ \cup \ \{(a \prec a') \mid (a \prec a') \in \mathcal{O}\} \ \cup \\ &\qquad \{(a \prec a') \mid a' \in N(f), \forall f \in ADD(a) \cap \psi\} \end{aligned}$$

- N_{init}: An initial mapping of fluents in the domain to an empty set:

$$\forall f \in F, N_{init}(f) = \emptyset$$

Algorithm 9: Partial Plan Context Generator

Input: ATPOP $\langle\langle \mathcal{A}, \mathcal{O}\rangle, \mathcal{C}\rangle$ and planning problem $\Pi = \langle F, O, I, G\rangle$
Output: Set of partial plan contexts, Ctx.

/* Ctx is the set of partial plan contexts to be returned */
$Ctx = \emptyset$;
/* Γ is a set of tuples containing a condition, POP, set of */
/* ordering constraints, and a mapping of fluents to actions */
$\Gamma = \{\langle \emptyset, \langle \mathcal{A}, \mathcal{O}\rangle, \emptyset, N_{init}\rangle\}$;
for $i = 1 \cdots |\mathcal{A}|$ **do**
  $\Gamma' = \emptyset$;
  foreach $\langle \psi, \langle \mathcal{A}_{\vdash}, \mathcal{O}_{\vdash}\rangle, \mathcal{O}_*, N\rangle \in \Gamma$ **do**
    foreach $a \in last(\langle \mathcal{A}_{\vdash}, \mathcal{O}_{\vdash}\rangle)$ **do**
      /* Compute the partial plan context */
      $\mathcal{A}_{\dashv}, \mathcal{O}_{\dashv} = \textsc{MakeContext}(\langle \mathcal{A}, \mathcal{O}\rangle, \mathcal{A}_{\vdash}, \mathcal{O}_*, a, \psi, N)$;
      $Ctx = Ctx \cup \{\langle \mathcal{A}_{\dashv}, \mathcal{O}_{\dashv}, \mathcal{R}(\psi, a), a\rangle\}$;
      /* Update the sets of actions needing a fluent. */
      $N' = N.copy()$;
      foreach $f \in ADD(a)$ **do**
        $N'(f) = \emptyset$;
      foreach $f \in PRE(a)$ **do**
        $N'(f) = N'(f) \cup \{a\}$;
      /* Update for the next iteration */
      $\Gamma' = \Gamma' \cup \{\langle \mathcal{R}(\psi, a), prefix(\langle \mathcal{A}_{\vdash}, \mathcal{O}_{\vdash}\rangle, a), \mathcal{O}_{\dashv}, N'\rangle\}$;
  $\Gamma = \Gamma'$;
return Ctx;

Mapping a fluent to the set of actions requiring it is essential for the temporal reasoning to respect an ordering that allows for every goal and precondition to be achieved. Without the additional ordering constraints, we may find a context that is both temporally and causally viable, but not viable in general (recall that we require a common linearization satisfying both forms of viability).

There are two fundamental differences between Algorithms 6 and 9: (1) rather than simply record the condition ψ and candidate action a, we also record the set of actions and ordering constraints to build a partial plan context, and (2) additional ordering constraints are computed through the use of MakeContext to ensure that, when establishing temporal viability, we reason about the correct linearization. Both modifications are primarily for bookkeeping and the soundness of the subsequent steps. Neither modification has a significant impact on the algorithm's performance.

We preserve key properties of Algorithm 6, and as such the following proposition follows from Theorem 6.

Proposition 2. *Every partial plan context,* $\langle \mathcal{A}_{\dashv}, \mathcal{O}_{\dashv}, \psi, a \rangle$*, produced by Algorithm 9 is causally viable with respect to* Π *and* s *iff* $\psi \subseteq s$*.*

Online, we can use the set of partial plan contexts returned by Algorithm 9 as is – iterating through the entire set to find those contexts that are causally viable for the observed state. However, it is far more efficient to store the partial plan contexts in a compact fashion as we did in Chapter 4. The data structure that we require is more complex than an ordered decision diagram, and we provide further details in Section 5.4.4.

5.4.3 Temporal Viability

Given the temporal constraints, for each partial plan context, Compiler determines temporal viability by proving consistency of a carefully constructed STN.

Definition 16 (Context-Specific STN)**.** With respect to the ATPOP $\langle \langle \mathcal{A}, \mathcal{O} \rangle, \mathcal{C} \rangle$ the *context-specific STN* (CSTN) of the partial plan context $ctx = \langle \mathcal{A}_{\dashv}, \mathcal{O}_{\dashv}, \psi, a^* \rangle$ is the STN $\langle E^{ctx}, C^{ctx} \rangle$ where E^{ctx} is induced by the events that appear in the set of constraints C^{ctx}, and C^{ctx} consists of the following three sets of simple temporal constraints:

1. Temporal constraints on the unexecuted actions in $\mathcal{A}_{\dashv}$:

$$\{[\epsilon, \infty]_{X_a, X_b} \mid (a \prec b) \in \mathcal{O}_{\dashv}\}$$

2. Past temporal constraints ending in $\mathcal{A}_{\dashv}$:

$$\{[l, u]_{X_a, X_b} \mid (\textbf{latest-before}\ b\ a\ l\ u) \in \mathcal{C}, b \in \mathcal{A}_{\dashv}\}$$

3. Future temporal constraints involving only $\mathcal{A}_{\dashv}$:

$$\{[l, u]_{X_a, X_b} \mid (\textbf{earliest-after}\ a\ b\ l\ u) \in \mathcal{C}, a, b \in \mathcal{A}_{\dashv}\}$$

A CSTN contains events corresponding to the scope of the set of simple temporal constraints *relevant* to the context – every other constraint from $\mathcal{C}$ can be safely ignored. If the CSTN is not temporally consistent, then there is no execution trace that can be extended with the context and be temporally viable. Compiler creates a CSTN

for every partial plan context and checks its consistency to ensure the context can be used online. Note that we ignore future constraints with a single action outside of $\mathcal{A}_{\dashv}$, because without knowing if the first action appears in the execution trace, the CSTN should not include the action. Below, we will deal with this situation specifically in the online setting.

Online, a consistent CSTN may or may not lead to a temporally viable partial plan context depending on the actual timing of *occurrences*. To enable a quick, online re-calculation of temporal viability, COMPILER stores the temporal windows between the events in the CSTN: the CSTN's distance graph.

Because we check the consistency of the CSTN using an all-pairs-shortest-path method, the shortest distance between any pair of events serves as the lower and upper bound on the relative timing between the events (cf. Section 5.2.1). For a distance graph $G = \langle V, E, w \rangle$, we denote the lower time bound between events X_1 and X_2 as $G.l(X_1, X_2) = -w((X_2, X_1))$. Similarly, we denote the upper time bound between events X_1 and X_2 as $G.u(X_1, X_2) = w((X_1, X_2))$. If X_1 represents the "start of time" in the CSTN, we will simplify $G.l(X_1, X_2)$ and $G.u(X_1, X_2)$ to just $G.l(X_2)$ and $G.u(X_2)$.

5.4.4 Representation

The first phase of the EXECUTOR finds the causally viable contexts given the state of the world. We could, in principle, iterate through the set of contexts returned by Algorithm 9, however this would be largely inefficient. Instead, we use a match tree representation to map the state of the world to the contexts that are causally viable.

Recall from Section 2.2.4 that in order to construct a match tree, we require a set of multi-valued variables V, a set of objects O, and a set $Pairs$ of $\langle p, o \rangle$ pairs for the mapping. We construct the three sets as follows:

- $V \stackrel{\text{def}}{=} F$ and $\forall v \in V, D(v) = \{true, false\}$: A fluent from the domain corresponds to a 2-valued variable.
- $O \stackrel{\text{def}}{=} C$: The set of contexts computed by Algorithm 9.
- $Pairs$: The set of contexts with their sufficient condition:

$$Pairs \stackrel{\text{def}}{=} \{\langle ctx.\psi, ctx \rangle \mid ctx \in Ctx\}$$

To construct the match tree representation for the set of contexts Ctx, we use the method MAKEMT($Pairs$, $\emptyset$). However, rather than using the heuristic described in

Section 2.2.4, we choose our variables based on the number of contexts in which they appear:

$$\textsc{PickVariable}(Pairs, Vars) = \max_{v \in Vars} |\{ctx \mid \langle \psi, ctx \rangle \in Pairs, v \in vars(\psi)\}|$$

Note that we do not need to prune the variables using $PairVars$ (i.e., those variables that do not appear in any pair in $Pairs$), because the heuristic maximizes the number of pairs in which a variable appears. The motivation behind this heuristic is to satisfy as many of the conditions as early as possible. This leads to line 1 of the MakeMT Algorithm being satisfied early on in the procedure. The result is a match tree G that we can use through ReadMT(G, s) to return the contexts that are causally viable for the current state of the world s. That is:

$$\textsc{ReadMT}(G, s) = \{ctx \mid ctx \in Ctx, s \models ctx.\psi\}$$

5.5 Temporally Executing Compiled ATPOPs

In this section, we describe the second component of TpopExec: Executor. The task of Executor is to determine the best course of action given the current situation. In our work, the current situation at runtime consists of the state of the world, the timing of actions executed so far, and the current time. Executor uses the information computed by Compiler to efficiently perform the task of deciding what to do next.

5.5.1 General Approach

Algorithm 10 shows the general procedure we use for online execution given (1) a match tree G mapping states to a set of contexts, (2) the state of the world s, (3) the current time t^*, and (4) the agent's execution trace $\mathcal{T}$.

The SortContexts method sorts the set of contexts based on their quality. Here, the quality of a context is measured as the total cost and ties are broken based on an approximation of the time expected to achieve the goal: minimum bound between the candidate action of a context and the goal action.

If line 4 of Algorithm 10 passes, the constructed STN is temporally consistent and there is a valid time window for the candidate action $ctx.a$ to be executed. Executor uses Algorithm 10 at every step to decide what to do next. The algorithm returns a

Algorithm 10: Online Execution

Input: Match tree of partial plan contexts G, current state of the world s, current time t^*, and the execution trace so far, $\mathcal{T}$.
Output: Action a and temporal window $[l, u]$ for execution.

```
/* Retrieve all of the causally viable contexts */
CV = ReadMT(G, s);
/* Sort the contexts based on quality */
SortContexts(CV);
/* Find the first temporally viable context */
foreach ctx ∈ CV do
    /* Check for temporal viability */
    if TemporallyViable(ctx, T, t*) then
        return ⟨ctx.a, [ctx.cstn.l(X_ctx.a), ctx.cstn.u(X_ctx.a)]⟩;
/* Otherwise, no viable context exists */
return no_viable_context;
```

tuple $\langle a, l, u \rangle$, and the agent has the choice to execute the action a at any time between the bounds of l and u.

5.5.2 Checking Temporal Viability

During execution, Executor has access to the current time, an execution trace, and the stored temporal windows for events that have occurred. To determine temporal viability, Executor uses the following three-step process for TemporallyViable$(ctx, \mathcal{T}, t^*)$:

1. If there is any past temporal constraint that is missing a starting action from the trace $\mathcal{T}$, return false. Formally, this occurs when the following holds:

$$\exists \ \textbf{(latest-before } b \ a \ l \ u\textbf{)} \in \mathcal{C}, \text{ such that}$$
$$a \notin ctx.\mathcal{A}_{\dashv} \ \wedge \ b \in ctx.\mathcal{A}_{\dashv} \ \wedge \ \forall((a_i, s_i), t_i) \in \mathcal{T}, a_i \neq a$$

2. If there are unresolved future constraints in the trace, rebuild the CSTN and recheck its temporal consistency. Return false if the network is not temporally consistent.

3. Simulate the execution of past events in the CSTN and return false if the current time t^* is outside of the temporal window for the candidate action $ctx.a$.

These steps are performed on a copy of the context's CSTN. For every unresolved future temporal constraint **(earliest-after** $a \ b \ l \ u$**)**, we have a set of *occurrences* that

are the cause for the constraint remaining unresolved (i.e., the *occurrences* containing a that have happened *after* the most recent *occurrence* containing the action b):

$$\textsc{Active}(\textbf{(earliest-after } a\ b\ l\ u\textbf{)}) = \{((a_i, s_i), t_i) \mid \textbf{(occ } a\ i\textbf{)} \wedge \forall j > i, \neg\textbf{(occ } b\ j\textbf{)}\}$$

If X_a does not already exist in the CSTN, then Executor adds event, X_a, corresponding to the *latest occurrence containing* a in $\textsc{Active}(\textbf{(earliest-after } a\ b\ l\ u\textbf{)})$ and includes the simple temporal constraint $[l, u]_{X_a, X_b}$. If there is more than one *occurrence* that serves as a reason for the unresolved constraint, Executor adds another event, X'_a, to the CSTN corresponding to the *earliest occurrence* containing a in the set $\textsc{Active}(\textbf{(earliest-after } a\ b\ l\ u\textbf{)})$. Again, we add the constraint $[l, u]_{X'_a, X_b}$. The remaining *occurrences* containing a can be ignored as they cannot further constrain the CSTN more than the earliest or latest – the extremes represent the tightest possible bounds.

After adding all of the required additional constraints to the network, Executor re-checks for consistency to ensure temporal viability. Methods exist for efficiently computing the temporal consistency after an incremental update of a temporal network (e.g., Gerevini *et al.* [41]), and we present further optimizations in the following section.

Using the standard dispatching algorithm for an STN (cf. Section 5.2.2), Executor tests whether or not a schedule exists for the actions in $A_{\dashv}$ that adheres to all of the temporal constraints and the execution trace. For every event X_a in the context's original CSTN where $a \notin A_{\dashv}$, we have a corresponding latest *occurrence* $((a, s_i), t_i) \in \mathcal{T}$. For the unresolved future temporal constraints, we also have an associated *occurrence* (or potentially two *occurrences*, as described above). Using the standard dispatching algorithm, Executor follows the order found in $\mathcal{T}$ to dispatch each event at the time already established, propagating the start times. If Executor must dispatch an event at a time outside of its temporal bounds, then the network is inconsistent (cf. Theorem 2 of Muscettola *et al.* [78]).

If the temporal windows remain non-empty, during the simulated dispatching (i.e., the lower bound never exceeds the upper bound), then the process ends with a temporal window for the event corresponding to the candidate action, a. This provides Executor with both a certificate that the CSTN is consistent, and an indication of what should be done: execute a within its temporal window. As mentioned, at the start of the TemporallyViable procedure, we copy the context's CSTN for the current situation. This copy is what is used on line 5 of Algorithm 10. Executor resets the CSTN for the context to a new copy every time it calls TemporallyViable.

5.5.3 Theoretical Results

Having described the general approach for online dispatching we verify the theoretical correctness of the procedure.

Theorem 9. *Given an ATPOP $\langle\langle \mathcal{A}, \mathcal{O}\rangle, \mathcal{C}\rangle$, valid partial trace $\mathcal{T}$, and partial plan context $\langle \mathcal{A}_{\dashv}, \mathcal{O}_{\dashv}, \psi, a\rangle$, the context is temporally viable iff* TemporallyViable$(ctx, \mathcal{T}, t^*)$ *holds.*

Proof. For the context to be temporally viable, $\langle \mathcal{A}_{\dashv}, \mathcal{O}_{\dashv}\rangle$ must have a linearization that corresponds to some trace $\mathcal{T}'$ such that $\mathcal{T} \cdot \mathcal{T}'$ satisfies every constraint in $\mathcal{C}$. The first set of constraints included in the construction of a CSTN ensures that any schedule of the CSTN follows a linearization of $\langle \mathcal{A}_{\dashv}, \mathcal{O}_{\dashv}\rangle$. The augmented CSTN includes a simple temporal constraint for every constraint in the ATPOP. The CSTN is consistent and dispatchable if and only if there is a schedule of the actions in $\mathcal{A}_{\dashv}$ that satisfies every constraint. We thus have a candidate for $\mathcal{T}'$ if and only if the context is temporally viable. □

We can now fully characterize how TpopExec leverages a partial plan context.

Theorem 10. *For a given planning problem Π, ATPOP $\langle\langle \mathcal{A}, \mathcal{O}\rangle, \mathcal{C}\rangle$, valid partial trace $\mathcal{T}$, state of the world s, and partial plan context $\langle \mathcal{A}_{\dashv}, \mathcal{O}_{\dashv}, \psi, a\rangle$, the partial plan context is viable iff (1) $\psi \subseteq s$, and (2) the context's CSTN can be dispatched given the observed start times of all relevant past events.*

Proof. Following from Proposition 2 and Theorem 9, we know that the context is both causally and temporally viable. What remains to be shown is that there is at least one linearization that suffices for both causal and temporal viability. The behaviour of Algorithm 9 leads to a stronger conclusion: every linearization that allows the context to be temporally viable suffices for the context to be causally viable.

The MakeContext method, in cooperation with the fluent mapping N, ensures that if an action achieves a fluent (thus removing it from the regressed formula) an ordering constraint is added to $\mathcal{O}_{\dashv}$. The result is that the CSTN will correctly order any pair of actions where the first must achieve a precondition of the second. Thus, every linearization that satisfies the temporal constraints in the CSTN will also satisfy the causal viability condition. □

Theorem 10 connects the partial plan contexts that Algorithm 9 generates with the desired conditions we have for a partial plan fragment to achieve the goal. Following Theorem 10, we have an analogous result to Corollary 1 from Chapter 4.

Corollary 2 (Execution Completeness)**.** *Given a planning problem* Π*, an ATPOP for* Π*,* $\langle\langle \mathcal{A}, \mathcal{O}\rangle, \mathcal{C}\rangle$*, a valid partial trace* $\mathcal{T}$*, and a state of the world* s*, if no partial plan context produced by Algorithm 9 is viable, then there is no timed sequence of actions that will both (1) correspond to a suffix of a linearization of* $\langle \mathcal{A}, \mathcal{O}\rangle$ *which satisfies the goal when executed in* s*, and (2) satisfy every temporal constraint in* $\mathcal{C}$ *when used to extend* $\mathcal{T}$*.*

Proof. This follows almost directly from Theorem 10 – the only further point to consider is if a suitable timed action sequence can exist when no partial plan context produced by Algorithm 9 is viable. In order for this to happen, the sequence of actions must achieve the goal from s and be a suffix of some linearization of $\langle \mathcal{A}, \mathcal{O}\rangle$. We would therefore consider the condition for this sequence to achieve the goal in Algorithm 9, along with a POP suffix that corresponds to the actions in the sequence. Because the temporal constraints remain as flexible as possible in the offline processing, the context will be temporally viable if the timed action sequence serves as an extension for $\mathcal{T}$ such that every constraint in $\mathcal{C}$ is satisfied. We are therefore guaranteed to have such a partial plan context produced by Algorithm 9. □

Corollary 2 essentially stipulates TPOPEXEC avoids replanning whenever possible – TPOPEXEC will recognize every situation where some partial plan fragment can achieve the goal while satisfying every temporal constraint.

In contrast with Chapter 4, the use of additional ordering constraints, while essential to Theorem 10, causes an increase in the number of unique contexts in the set Γ. This results in a larger storage requirement for the set of contexts. With TPOPEXEC, we have the following complexity results:

Proposition 3 (Viability Complexity)**.**

1. *The complexity of computing the causally viable contexts online is at worst linear in the number of contexts.*

2. *The complexity of determining temporal viability is at worst polynomial in the size of the CSTN.*

Determining causal viability is at worst linear as technically every context may be applicable (e.g., in a state where everything is true). In practice, however, the average complexity is much smaller – typically it is linear in the size of the relevant portion of the current state. The complexity of determining if a context is temporally viable is polynomial because at worst we must run a temporal consistency check on the context's CSTN. However, we have identified many heuristic checks that successfully determine if

the CSTN is temporally consistent in the overwhelming majority of situations. We detail these improvements in the following section.

The number of contexts and temporal networks may pose a problem if stored naïvely. However, by leveraging the commonality between the contexts and their temporal networks, we are able to drastically reduce the overall storage required to store both the state and temporal information compared to the method presented in Chapter 4.

5.5.4 Optimizations

We have presented the computational machinery to enable TpopExec to select both a subsequent action and the timing of its execution. Following Theorem 10, as long as a suffix of some linearization of the POP can achieve the goal while satisfying all temporal constraints, TpopExec will eventually achieve the goal. Executor filters first based on causal viability and then discards the contexts which are not temporally viable, because determining temporal viability requires some amount of reasoning online. In this section, we present some improvements on the described approach to improve the efficiency of online reasoning.

Relevant CSTN Events

When storing the constructed CSTN for a partial plan context, not every event needs to be included. We refer to the events corresponding to actions outside of $\mathcal{A}_{\dashv}$ as past events – those that appear at the start of a past temporal constraint will necessarily have already occurred. Similarly, we refer to the events corresponding to actions in $\mathcal{A}_{\dashv}$ as future events – using this partial plan context presumes that every action in $\mathcal{A}_{\dashv}$ will be executed. To improve on the storage and reasoning required, we store only the past events for the past temporal constraints and future events for the future temporal constraints, along with the events for the candidate action and initial state:

- Events for starting actions from past temporal constraints not appearing in $\mathcal{A}_{\dashv}$:

$$\{X_a \mid (\textbf{latest-before } b\ a\ l\ u) \in \mathcal{C}, a \notin \mathcal{A}_{\dashv}, b \in \mathcal{A}_{\dashv}\}$$

- Events for endpoint actions from future temporal constraints appearing in $\mathcal{A}_{\dashv}$:

$$\{X_b \mid (\textbf{earliest-after } a\ b\ l\ u) \in \mathcal{C}, b \in \mathcal{A}_{\dashv}\}$$

- The events corresponding to the candidate action and initial state: X_a and X_{a_I}.

We refer to the subset of events, and all of the temporal windows between them, as the *relevant CSTN*. Compiler only stores the relevant CSTN, but the original consistency check and computation of the temporal windows are computed using the full CSTN. The procedure described in Section 5.5.2 proceeds normally – every unresolved future constraint will still have an ending action in the stored CSTN and the simulation of past events remains unchanged. For the latter, the simulation is over the same set of events, and so there is little change. For the first step, however, we are checking the consistency of a subset of the original CSTN when we encounter an unresolved future temporal constraint.

Theorem 11. *Using the relevant CSTN produces the same temporal windows as the full CSTN for the candidate action of a partial plan context.*

Proof. The process of re-running the consistency check will change the temporal window for the candidate action only when a new shortest path is introduced by the unresolved future temporal constraints. Because we have retained the endpoints for every edge that can be tightened in the d-graph, along with the shortest path between any pair of events, we have sufficient information to reproduce the shortest paths between the event corresponding to the initial state action and the event corresponding to the candidate action (i.e., the temporal windows for the candidate action $ctx.a$).

To see why this is so, consider what such a 'shortest path' will look like in the original CSTN: a sequence of events starting with X_{a_I} and ending in $X_{ctx.a}$. Any edge introduced with a higher weight has no effect on the shortest path, so without loss of generality we consider only those edges that have a new smaller weight due to a future temporal constraint. Assume that these edges are denoted as $X_a \to X_{a'}$, $X_b \to X_{b'}$, etc. Any new shortest path from X_{a_I} to $X_{ctx.a}$ must be of the following form:

$$X_{a_I} \rightsquigarrow X_a \to X_{a'} \rightsquigarrow X_b \to X_{b'} \rightsquigarrow \cdots \rightsquigarrow X_{ctx.a}$$

We use $\rightsquigarrow$ to signify a sequence of events with edge weights that have not changed from the future temporal constraints. Note that the distance of every $X_{a'} \rightsquigarrow X_b$ sequence is retained in the relevant CSTN through the shortest path information between X_a and $X_{a'}$. Thus, every intermediate event is not required to compute the shortest path between X_{a_I} and $X_{ctx.a}$. A similar argument can be made for the opposite direction (i.e., shortest path from the candidate action's event to the initial state event), giving us the correct temporal window for the candidate action using only the relevant CSTN. □

Sufficient Temporal Consistency Checks

Instead of performing a full consistency check for testing temporal viability in the presence of unresolved future temporal constraints, Executor evaluates a number of necessary or sufficient conditions first. Executor uses a full consistency check only when the more efficient checks fail.

The first technique is to perform a dry run of the simulation, which will propagate temporal windows to the candidate action as well as every endpoint for a future temporal constraint. After the simulation, we check to see if the future temporal constraint would restrict the timing of the endpoint action in any way. If the bounds are not tightened, there is no need to consider that constraint for consistency (i.e., the network is already sufficiently tight to satisfy the future temporal constraint).

The second technique is to ensure that the upper bound on the timing of the candidate action is always larger than the current time. This is done throughout the simulated dispatching (when bounds are being propagated) and after the consistency is re-checked.

Finally, when there is a single unresolved future temporal constraint, the bounds can be directly propagated to the candidate action. This is because there are just two possibilities to reduce a shortest path, and checking them manually allows us to avoid recomputing the consistency of the relevant CSTN.

5.6 Temporal Relevance

The prevailing motivation for the work in this chapter is to enable the robust execution of a plan with temporal constraints in an environment that can unexpectedly change in various ways. One striking example of where an agent must re-execute actions is the behaviour of an agent such as a home robotic assistant that is expected to continually perform actions until manual intervention dictates otherwise. In such scenarios, goals frequently become unachieved (e.g., the floor repeatedly becomes dirty), and it can be detrimental to performance to consider the entire history of the agent's action sequence. As described in Sections 5.4 and 5.5, we elect to determine what is *relevant* for the required temporal reasoning. The notion of ordering relevance we presented in Section 3.3 improves the flexibility of a plan during generation and the notion of state relevance presented in Section 4.3 improves the robustness of plan execution. Here, temporal relevance improves the efficiency of plan execution in the presence of temporal constraints. Formally, we use the following notion of temporal relevance:

Definition 17 (Temporal Relevance)**.** Given an ATPOP $P = \langle\langle \mathcal{A}, \mathcal{O}\rangle, \mathcal{C}\rangle$, partial plan

context $C = \langle \mathcal{A}_{\dashv}, \mathcal{O}_{\dashv}, \psi, a \rangle$, and execution trace $\mathcal{T}$, the possibly non-contiguous subsequence $\mathcal{T}'$ of $\mathcal{T}$ is a *temporally relevant execution sub-trace* with respect to P, C, and $\mathcal{T}$ iff the temporal viability of C can be determined using only P, C, and $\mathcal{T}'$.

We use the notion of temporal relevance when deciding what events to use for the simulated dispatching to test for temporal viability in Executor. Typical schedule dispatching techniques assume that every event will be executed exactly once, and so they do not consider looking at a relevant subset for reasoning. One exception to this trend is the work on monitoring temporal logics, such as TLTL [7]. Their approach constructs various monitors in the form of event-clock automata, which will implicitly keep track of only the relevant past events.

5.7 Evaluation

The core contribution of this work is the ability to execute a plan in a world that can change in unpredicted ways, while reasoning about ongoing causal and temporal viability. Crucially, we accomplish this while avoiding unnecessary replanning, rescheduling, or plan repair. Building on the work in Chapter 4, TPOPExec is able to continue executing a POP in exponentially many more states than traditional approaches that execute actions according to a prescribed ordering. In particular, the improvement in robustness that Chapter 4 introduced through plan generalization techniques apply equally well to a temporally constrained plan through the use of Algorithm 9.

Standard approaches to schedule dispatching, such as Muscettola *et al.* [78], are blind to causal viability and the conditions for the executability of actions. Such approaches will succeed only on problems that do not experience obstructive change. Our evaluation focuses on TPOPExec's robustness and ability to avoid replanning. We also evaluate the general properties of TPOPExec's behaviour. Analytically, TPOPExec has the potential to avoid replanning in up to an exponential number (in the size of the state) of situations. In practice, this potential is reduced by the number of fluents that are relevant to plan validity and the extent to which a valid plan is causally and temporally constrained.

In this section, we describe our implementation of the TPOPExec procedures and present an empirical evaluation of its capabilities.

5.7.1 Implementation and Experimental Setup

The first experiment (Section 5.7.2) demonstrates the increased capabilities of our approach over restricted forms of our method that improve on existing execution strategies

(i.e., STN dispatching), and the second experiment (Section 5.7.3) examines the amount of replanning TpopExec avoids. The third experiment (Section 5.7.4) investigates the improvement that our pre-processing introduces by avoiding expensive temporal reasoning online. Finally, we provide some details on the behaviour of the system in Section 5.7.5. TpopExec is written in Python, and we conducted the experiments on a Linux desktop with a 3.0GHz processor.

The benchmarks from the International Planning Competition[1] (IPC) lack a combination of causal requirements and complex temporal constraints. As such, we tested our implementation on an expanded version of the CA domain that serves to challenge the causal and temporal reasoning, and is representative of what we might find in the real world. In total, there are 19 actions in the plan (11 durative), 8 past temporal constraints, and 4 future temporal constraints. The ATPOP has 18 ordering constraints which result in a total of 49,140 linearizations. A full description of the domain can be found in Appendix B.4.

The types of un-modelled dynamics include children becoming hungry, laundry being soiled, etc. In addition to being modelled after real-world temporal requirements, the constraints were designed to pose a challenge for TpopExec. For example, in the CA domain there are ample opportunities for a context to be causally viable but not temporally viable – the actions in the context can achieve the goal, but not without violating some temporal constraint. Such situations challenge the temporal reasoning aspects of TpopExec to find the most appropriate context.

5.7.2 Rate of Success

In Chapter 4 we demonstrated the robustness afforded to an agent that creates a policy from a POP. Using our technique for POP generalization, an agent can continue executing in exponentially more states than the original plan would be applicable in. The work in this chapter inherits this robust behaviour, plus we can further leverage the flexibility afforded by our temporal reasoning. For evaluation, we simulate our CA agent in a world where fluents change unexpectedly in both positive and negative ways (i.e., adding and deleting fluents from the state). TpopExec fails when Executor determines that the goal is no longer causally and temporally achievable.

The level of variability in the world is set using parameter α: a value of 0 corresponds to no changes whatsoever and a value of 1 corresponds to significant unpredictable change (at least one fluent changes after every action with 99.998% probability). The probability

[1] `http://ipc.icaps-conference.org/`

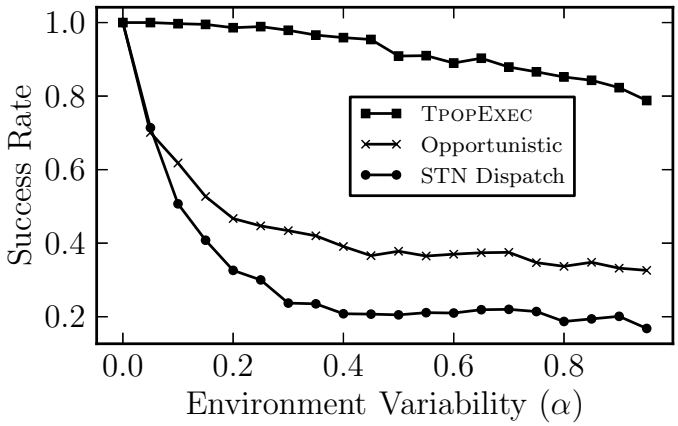


Figure 5.4: The success rate of the three approaches over a range of environment dynamics, both good and bad.

of an individual fluent changing ranges linearly from 0 to roughly 0.3 depending on the value of α. More details on how the probabilities are computed can be found in Appendix B.4. For 20 different values of α, we ran 1000 trials for each approach. The proportion of successful trials is referred to as the *success rate*.

The method from Chapter 4 cannot succeed for the CA example, as the method is oblivious to the temporal constraints found in the domain. Conversely, ignoring the causal requirements and simply dispatching the ATPOP one action after another mirrors STN dispatching, which we argued above would be unsuccessful in most instances. Nonetheless, we test this approach to verify our intuition and evaluate the level of environmental variability that an STN dispatching algorithm can handle.

We also present an *Opportunistic* version of TpopExec that we restricted to execute an action at most once. It can, however, skip actions if positive changes allow. Figure 5.4 shows the success rate for all three approaches for a given value of α.

TpopExec consistently outperforms both the ablated version and STN dispatching by a substantial margin – successfully executing the plan in more than 80% of the instances for almost all values of α and in total succeeding in over 92% of the simulations compared to just over 30% for the STN approach. Having the opportunity to re-execute plan fragments that are required again, while adhering to the imposed temporal constraints, provides us with a distinct advantage. The ablated version, while heavily restricted, still outperforms the STN dispatching for the majority of α-values, solving roughly twice as many instances.

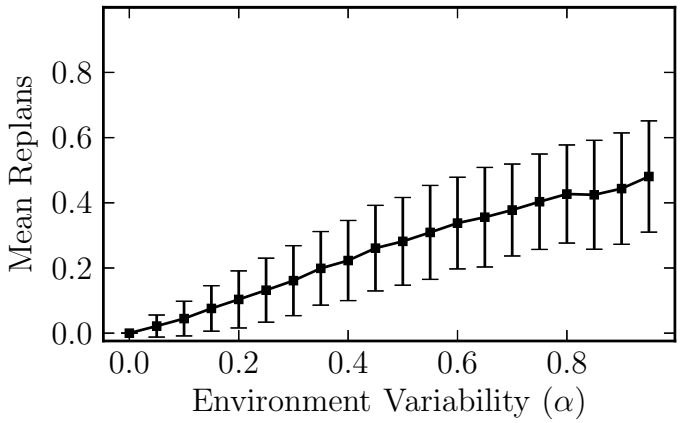


Figure 5.5: The fraction of replans avoided during execution over a range of destructive environment dynamics (mean and standard deviation).

5.7.3 Replan Avoidance

The key motivation for our approach is to avoid replanning or plan repair if at all possible. To this end, we evaluate how often TpopExec avoids replanning during execution in the CA domain. If other systems that replan online are able to replan quickly enough, then their execution behaviour would match ours. Every replan, however, requires solving a PSPACE problem which is what we avoid.

Similar to the previous experiment, we evaluate with respect to a range of environment dynamics (the α parameter). However, to properly gauge the need for replanning, we allow for only negative changes to the world: fluents are randomly made false. We count the number of times during execution that TpopExec would be forced to replan if it had not compiled the ATPOP, and we consider only those runs where TpopExec reaches the goal. Figure 5.5 shows the mean replan rate for a given α-value, normalized by a theoretical maximum number of replans.

We find that the number of avoided replans increases linearly with the increase variability. Due to the temporal constraints on the length of the day, and the length of some durative actions, there is a theoretical limit of roughly 20 replans required for the dynamics we introduce. In situations where TpopExec must operate over a larger time frame, we would expect the potential for replan avoidance to grow.

5.7.4 Preprocessing Impact

An alternative to the preprocessing we do is to use our ContextGenerator algorithm offline, and when online repeatedly re-run the all-pairs-shortest-path procedure to main-

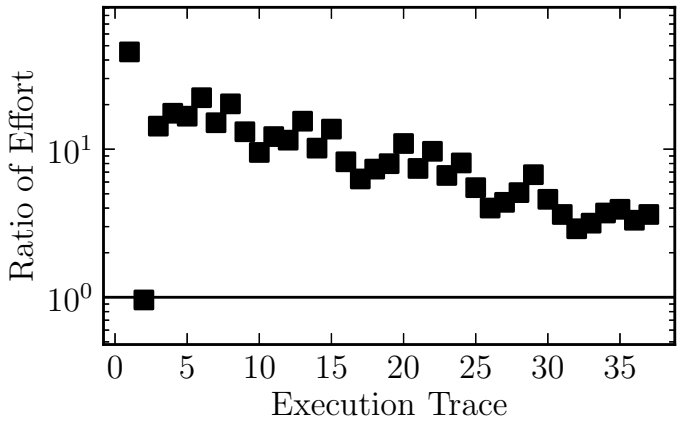


Figure 5.6: Amount of time required online for temporal reasoning fully at each step in the execution trace divided by the time required when using the preprocessed information.

tain consistent action timing. This approach avoids some of the intricacy and memory overhead of our offline temporal reasoning. It also has the same observed execution behaviour as our full approach in the experiment reported above, ignoring the time per execution step.

To evaluate the impact of our preprocessing techniques, we divide the computation time for the alternative approach at every step in an execution trace by the time for our method. The ratio between the two over a representative execution trace for the CA domain is shown in Figure 5.6. The uncertainty that we introduced randomly negated fluents with an environmental variability of $\alpha = 0.5$.

Each point above the $y = 1$ line is a step in the execution trace where preprocessing has an advantage. One outlier results from an open future temporal constraint. The conditions we employ to test for temporal viability were not sufficient and the agent had to re-run the all-pairs shortest-path algorithm to ensure temporal viability. The slight penalty we see (i.e., ratio less than 1) is due to the overhead associated with the failed testing for the sufficient conditions of temporal viability. In general, we find an decrease in the time required between a factor of 3 to 12 with the improvement being far more pronounced early in the plan where the number of actions in the partial plan context is larger.

5.7.5 System Behaviour

Profiling Executor, we found that 35% of the time was spent determining whether or not contexts were causally viable and roughly 60% of the time was spent determining if contexts were temporally viable. The remaining time was used for bookkeeping and

data-structure updates for the simulation. Compiler spent the vast majority of its time checking the consistency for the CSTN of each of the 306 partial plan contexts. There is substantial commonality between CSTNs of similar contexts, and a potential optimization is to reuse the computation results.

An average of 19 temporal windows were required for every CSTN. However, the data-structure used for computing the causally viable contexts is the memory bottleneck. Using the match tree representation detailed in Section 5.4.4, we were able to reduce the memory requirements for our causal information by a factor of four compared to the OADD technique introduced in Chapter 4, while our total footprint (causal plus temporal information) used roughly half the memory of storing the causal information with the previous representation.

5.8 Discussion

In this chapter we presented a system for compiling and robustly executing a plan that is augmented with temporal constraints. In the face of unexpected changes in the world, our system can select from a large number of valid plan fragments that are consistent with the temporal constraints, repeating parts of a plan or omitting actions, as necessary. The primary objective of our work is to minimize the need for replanning while accommodating the temporal constraints in the face of unexpected changes. To accommodate this flexibility, we introduced temporal constraints over actions and fluents, formalizing their semantics with respect to the execution trace.

During execution, our system TpopExec identifies the partial plans computed offline that can achieve the goal while satisfying all of the temporal constraints. To choose an action for execution, TpopExec selects one at the start of the best quality partial plan identified as being both causally and temporally viable. A key contributor to the efficiency of TpopExec's online execution strategy is to only focus on the relevant execution history: instead of considering the entire execution history to determine the continued satisfiability of a temporal constraint, TpopExec identifies a far smaller subset of the execution history that is relevant for the constraint to be satisfied.

We aim to address two fundamental limitations with previous related work: 1) temporal reasoning and schedule dispatching techniques typically do not consider the state of the world, and 2) execution monitoring schemes for planning problems that allow multiple action occurrences typically do not allow for temporal constraints to be defined. The temporal constraints that we introduce are an essential ingredient for the synthesis of plan execution and schedule dispatching techniques when the environment can change

in unexpected ways. They also elucidate the need for referring to both state and actions as integral parts of a temporal constraint.

Here, we discuss work related to ours and conclude with some general discussion surrounding our approach and potential avenues for future work.

5.8.1 Related Work

In addition to the literature our work directly builds upon, there are several related areas of research. Most approaches to executing plans with complex temporal constraints assume that an action in a plan will be executed once: an action may appear multiple times in a plan, but each plan appearance corresponds to exactly one action-occurrence. IxTeT-eXeC executes actions from a temporally restricted POP and monitors the sufficient conditions for continued causal and temporal viability, replanning when they fail to hold [68]. Similarly, the Pike system executes a temporally-restricted POP while continuously monitoring a weaker set of conditions for temporal viability [70]. TpopExec can be seen as an improved executor that attempts to avoid replanning.

The Drake system focuses primarily on executing a plan with complex temporal constraints [20]. While it can choose not to execute an action if non-execution is explicitly included as part of a complex temporal constraint, Drake does not represent or reason about causal validity. One avenue we hope to pursue involves using the Drake temporal reasoning in place of our CSTNs. Doing so would allow TpopExec to handle more expressive constraints, such as the **sometime-before** constraint.

There is a large body of research on plan execution monitoring (EM) (e.g., Pettersson [82], Fritz and McIlraith [38], and Doherty *et al.* [30]). Loosely described, EM involves continually ensuring that a set of formulae holds and has a chance to continue holding, without knowing what the sequence of events in the future will be. In our work, we wish to identify not only whether or not the execution trace thus far is valid, but additionally how to achieve the goal while remaining valid. Fritz and McIlraith present a notable exception to the typical EM techniques in which the continued optimality of a plan being executed is monitored [38]. Some systems, such as the work of Doherty *et al.* [30], monitor temporal constraints, but many do not. Typically they focus on the feasibility of just one partial sequential plan and resort to replanning when any condition is violated. The work of Yu and Williams, on the other hand, focuses on iteratively relaxing an over-constrained temporal problem until a solution is feasible [106].

There have been a number of temporal logics introduced for monitoring plans and schedules (e.g., Koymans [63] and Kvarnström *et al.* [65]). The most related to our work is

TLTL [7]: it uses timed words at the core of its specification and provides a syntax capable of expressing the temporal constraints available to an ATPOP. They do not, however, include a mechanism for deciding what to do next. The constraints **(latest-before** b a l u**)** and **(earliest-after** a b l u**)** can be respectively specified in TLTL as: $\square(b \rightarrow \triangleleft_a \in [l, u])$ and $\square(a \rightarrow \triangleright_b \in [l, u])$. It is of interest to consider how we might expand our constraint specification language to handle all of TLTL or a similar logic.

5.8.2 Conclusion

We demonstrated our methodology through a prototype implementation and a series of experiments to test its robustness and flexibility. In a simulated uncertain environment for a real-world inspired domain, TpopExec is able to consistently avoid replanning when some portion of the initial plan remains valid, and it achieves the goal in 92% of the trials while the standard STN dispatching technique succeeded only 30% of the time.

There may exist a trade-off between the time saved by avoiding replanning and the quality of a new plan that could be found. In this work, we assume that replanning should be avoided if at all possible, but we hope to consider this trade-off in future work. As the contributions of TpopExec can be seen as complementary to many existing execution monitoring systems (e.g., IxTeT-eXeC or Kirk [68, 57]), we plan on incorporating our techniques into a larger system for wider application.

We hope to pursue several other extensions of our work. First, we expect that it will be straightforward to extend the temporal reasoning to allow for concurrent actions, however, it will be necessary to include a model of concurrency in the causal domain (e.g., Boutilier and Brafman [13]). Another aspect we wish to explore is that of temporal uncertainty. Taking uncertain durations into account, the agent's behaviour can mirror that of *dynamic controllability* [97]. The high-level idea would be for the agent to wait for either an uncontrollable action to occur or for a particular duration of time to pass before committing to an action. Finally, we will work towards extending the richness of our temporal constraint specification, as achieving a full logic specification would greatly increase the expressivity available to the user.

Chapter 6

Efficient Non-deterministic Planning

6.1 Introduction

In the previous chapters we demonstrated how we can leverage relevance to generate and flexibly execute plans. In this chapter, we shift our focus to generating plans where the uncertainty is captured as non-deterministic action effects. We rely heavily on state relevance to improve plan generation in a non-deterministic setting, and in doing so we make several contributions to the state of the art in non-deterministic planning. Here, a solution to a non-deterministic planning problem is a policy that maps the state of the world to an action for execution. We introduce flexibility by means of a compact policy representation and provide robustness by employing a procedure that iteratively computes solutions with a greater likelihood of success.

When an agent executes a plan, there may be aspects of the environment over which the agent has no control. If we have some idea of what the potential outcomes may be, we can model the uncertainty through the use of non-deterministic actions in the model. In the planning community, non-deterministic action outcomes have been studied in the area of fully observable non-deterministic (FOND) planning (e.g., Daniele *et al.* [23]). The related field of probabilistic planning focuses on actions with outcomes that occur with some probability [84]. The former deals primarily with finding a policy or a plan conditional on the outcome of the actions, while the latter typically focuses on either maximizing the likelihood of achieving the goal or maximizing the overall reward given by actions in the plan. While the settings and planning objectives are formulated differently, the two forms of planning share a number of characteristics.

We focus on a class of solution techniques for FOND planning problems that involves determinizing the actions by creating a separate action for each non-deterministic outcome and using a classical planner to find a solution (e.g., the planners by Kuter *et al.* [64]

and Fu *et al.* [39]). The planner then enumerates the state space by following the policy that maps complete states in the classical plan to the next action in the plan. When the planner encounters a non-deterministic action, it considers every possible successor state. The planner treats any state that it has not seen, as the initial state for a new planning problem. In a similar sense, in a probabilistic setting one well-known approach to achieving a goal condition with high probability is to plan with a determinized version of the domain and to replan online only when the planner encounters a state that it has not yet seen (e.g., FF-Replan [101]). While there is a range of probabilistic planning techniques for a variety of probabilistic planning formalisms, using replanning via determinization is the most related to our work.

Both approaches execute the policy and repair the missing parts of the policy based on the action outcome – either in offline simulation or online execution. Typically, both approaches use the entire state to represent and reason about the plan, which is often far too detailed. A more succinct policy should map only the relevant portion of the state to the actions. This form of succinct policy representation is one example of how we leverage state relevance to efficiently construct and refine a policy. Doing so results in a policy that is far more robust than policies produced by previous approaches.

In this chapter we present our planner, PRP, which incorporates both FOND and probabilistic planning techniques to construct a solution to a FOND planning problem. If no such solution exists, PRP returns the policy with the highest quality it is able to find. Here, the quality of a policy is measured as the simulated likelihood of achieving the goal with the policy, assuming that every action outcome is equally probable. Online, our planner behaves similarly to the online replanning approaches for probabilistic planning. If we restrict the time for the offline phase, the planner uses the policy with the highest quality found within the time limit. The strength of this approach stems from our focus on only those parts of the state that are relevant – in many real-world problems only a small subset of the state plays a role in the successful execution of a plan. PRP includes several novel techniques for non-deterministic planning based on state relevance, and we demonstrate how these improvements allow our planner to outperform the state-of-the-art techniques in both FOND and probabilistic planning.

PRP generates policies up to several orders of magnitude faster and several orders of magnitude smaller than the previous state-of-the-art FOND planner, FIP [39]. When compared to online replanning approaches inspired by probabilistic planning techniques, we find that PRP achieves the goal with far fewer actions than the method employed by FF-Replan [101]. Further, when we consider "probabilistically interesting" domains that have the potential for deadends [71], we find that PRP scales better than the state of the

art in online replanning tailored to handle domains with deadends, FF-Hindsight+ [103]. PRP solves problems with a perfect success rate while FF-Hindsight+ only succeeds in as low as 65% of the simulations.

6.1.1 Contributions

- We introduce a state-of-the-art planner for fully observable non-deterministic planning problems: PRP. We demonstrate empirically that our system is able to compute robust solutions exponentially faster than previous methods and the resulting policies are exponentially smaller.

- We develop a suite of principled methods to leverage the relevant parts of the state of the world in non-deterministic planning. These methods include:

  1. Targeted replanning when the incumbent solution is insufficient.
  2. A compact and robust representation of the solution.
  3. Retaining only the relevant portion of dead-end states for efficient dead-end avoidance.
  4. Effective stopping conditions for PRP's simulation loop based on state relevance.

  We further prove the correctness of our algorithm in the presence of all the improvements made to the general procedure.

- We present a method for using non-deterministic planning in an any-time setting. Effectively, this gives PRP the flexibility to operate at any point in the spectrum between offline and online planning. During the offline phase, the incumbent solution is repeatedly improved until PRP computes a policy that handles every state reachable by the policy (if one exists), or PRP reaches a specified time-out.

- In addition to evaluating our techniques on non-deterministic domains, we demonstrate the effectiveness of our techniques in challenging domains with probabilistic action outcomes. When a solution exists that works with 100% probability, PRP is able to compute it and can do so far faster than existing approaches for probabilistic planning. If no such solution exists, PRP will still generate a policy, but there is no guarantee on the policy's quality.

The work in this chapter is based on a publication at ICAPS'12 [76]. The algorithms have been expanded on, and an illustrative example was added to many of the sections.

We additionally characterize how our approach can be used for strong planning and include an evaluation of the impact various parameters have on the proposed planner.

6.1.2 Organization

We start this chapter with a review of the required background and notation for non-deterministic planning. In Section 6.3 we present the overall approach for our planner. In Section 6.4 we describe our handling of dead-end states and in Section 6.5 we describe our technique for targeted replanning. We present our method for effectively limiting the required simulation in Section 6.6, and summarize a key component of the overall approach in Section 6.7. We outline how our approach can be applied to strong planning and online planning in Sections 6.8 and 6.9 respectively. Finally, in Section 6.10 we present our experimental evaluation of the PRP system and conclude with a discussion of related work and summary in Section 6.11.

6.2 Preliminaries

In this section we provide the notation used for FOND Planning Problems that use a multi-valued variable representation for the state of the world.

6.2.1 Non-Deterministic SAS+

For this chapter, we adopt the notation of Mattmuller *et al.* [72] for non-deterministic SAS$^+$ planning problems. A SAS$^+$ FOND planning task is a tuple $\Pi = \langle \mathcal{V}, s_0, s_*, \mathcal{A} \rangle$. $\mathcal{V}$ is a finite set of variables $v \in \mathcal{V}$, each having the finite domain D_v. We use D_v^+ to denote the extended domain of v that includes the value $\perp$ signifying the value of v is undefined.

A *partial state* is a function s that maps a variable $v \in \mathcal{V}$ to a value in D_v^+. If $s(v) \neq \perp$ then v is *defined* in s, and if every variable $v \in \mathcal{V}$ is defined for s, then s is a *complete state*. We will reuse the $vars(s)$ notation from Section 2.2.4 to signify the variables that are defined in s. The initial state s_0 of a SAS$^+$ FOND planning task is a complete state, while the goal state s_* is a partial state. Due to the fact that we use partial states frequently throughout this chapter, we introduce further notation to simplify the discussion:

- Entailment ($\models$): A partial state s entails another partial state s', denoted as $s \models s'$,

if and only if the following holds:

$$\forall v \in vars(s'), s(v) = s'(v)$$

- Consistency ($\approx$): Two partial states s and s' are said to be *consistent* with one another, denoted $s \approx s'$, if and only if the following holds:

$$\forall v \in \mathcal{V},\ \ s(v) = s'(v)\ \ \vee\ \ s(v) = \bot\ \ \vee\ \ s'(v) = \bot$$

- Updating ($\oplus$): The *updated* partial state obtained from applying partial state s' to partial state s, denoted as $s \oplus s'$, is the partial state s'' where,

$$s''(v) = \begin{cases} s'(v) & \text{if } v \text{ is defined for } s' \\ s(v) & \text{otherwise} \end{cases}$$

Note that s and s' need not be consistent for $s \oplus s'$ to be well-formed, and $\oplus$ is not commutative.

The final component of a planning task is the set of actions $\mathcal{A}$. Each action is made up of two parts: Pre_a, a partial state that describes the condition under which a may be executed; and Eff_a, a finite set of partial states that describe the possible outcomes of the action. If Eff_a contains multiple elements, then the agent is not able to choose which effect will take place.

An action a is *applicable* in state s if and only if $s \models Pre_a$ and a is *possibly applicable* in s if and only if $s \approx Pre_a$. We require the unconventional notion of possible applicability because our policy will be represented as a mapping of partial states to actions, rather than complete states. The *progression* of a partial state s, with respect to an action a and selected non-deterministic effect $e \in \mathit{Eff}_a$, denoted $Prog(s, a, e)$, is defined as follows:

$$Prog(s, a, e) \stackrel{\text{def}}{=} \begin{cases} (s \oplus Pre_a) \oplus e & \text{when } a \text{ is (possibly) applicable in } s \\ \textit{undefined} & \text{otherwise} \end{cases}$$

We say that a partial state s *can be regressed* through a with effect $e \in \mathit{Eff}_a$ if and only if $e \approx s$. The *regression* of partial state s with respect to an action a and effect $e \in \mathit{Eff}_a$, denoted $Regr(s, a, e)$, is the partial state s' defined for variable $v \in \mathcal{V}$ as follows:

$$s'(v) = \begin{cases} Pre_a(v) & \text{if } Pre_a(v) \neq \bot \\ \bot & \text{if } Pre_a(v) = \bot \text{ and } e(v) = s(v) \\ s(v) & \text{otherwise} \end{cases}$$

$Regr(s, a, e)$ is undefined if s cannot be regressed through a with e.

We assume that we are given a non-deterministic planning problem as a PDDL file with "oneof" clauses in the action effects, as is the case with FOND planning domains [15].[1] We convert the domains to a non-deterministic SAS$^+$ formalism using a modified version of the PDDL-to-SAS$^+$ translation algorithm [49].

6.2.2 FOND Planning

Unlike classical planning, where a solution is a sequence of actions, a solution to a FOND planning task is a policy that maps a state to an appropriate action such that the agent eventually reaches the goal. A policy is *closed* if it returns an action for every non-goal state that a policy reaches and a state s is said to be *reachable* by a policy if there is a chance that following the policy leads the agent to s. When the agent executes a non-deterministic action the effect is randomly chosen, so a closed policy must handle every possible outcome of an action it returns. There are three primary variations of plans for a FOND planning problem [23]: *weak*, *strong*, and *strong cyclic*.

Definition 18 (Weak Plan)**.** A *weak plan* is a policy that achieves the goal with at least one possible set of outcomes for the non-deterministic actions.

A weak plan may be as simple as a sequence of actions that achieves the goal with a particular setting of non-deterministic action outcomes. The policy for a weak plan need not be closed.

Definition 19 (Strong Plan)**.** A *strong plan* is a closed policy that achieves the goal and never visits the same state twice.

A strong plan provides a guarantee on the maximum number of steps to achieve the goal but is often too restrictive. For example, a strong plan cannot contain an action that can fail with no effect on the state.

Definition 20 (Strong Cyclic Plan)**.** A *strong cyclic plan* is a closed policy where the goal is reachable from every state reachable using the policy.

[1] We convert probabilistic planning domains to the FOND format by ignoring the probabilities.

A strong cyclic plan guarantees that the agent eventually reaches the goal, but does not guarantee that the agent can do so in a fixed number of steps. We are primarily interested in computing strong cyclic plans, as they are guaranteed to achieve the goal under an assumption of fairness – every outcome of a non-deterministic action will occur infinitely often if the action is executed infinitely often. There are a few approaches that produce a strong cyclic plan and we focus on one that enumerates weak plans until it creates a strong cyclic plan. We achieve this through a determinization of the domain.

Definition 21 (Determinization)**.** A *determinization* of a SAS$^+$ FOND planning task $\langle \mathcal{V}, s_0, s_*, \mathcal{A} \rangle$ is a planning task $\langle \mathcal{V}, s_0, s_*, \mathcal{A}' \rangle$ where $\mathcal{A}'$ is a modification of $\mathcal{A}$ such that every action in $\mathcal{A}'$ is deterministic. The *single outcome* determinization creates $\mathcal{A}'$ by removing all but one outcome from every action in $\mathcal{A}$. The *all-outcomes* determinization creates $\mathcal{A}'$ by replacing each non-deterministic action in $\mathcal{A}$ with a set of a new actions corresponding to each non-deterministic outcome.

There are further determinization techniques that have been studied (e.g., time-dependent [103] and layered [56]), but we focus on the all-outcomes determinization. Finding a classical plan for a determinization provides a weak plan for the non-deterministic planning task. Fu *et al.* [39] demonstrated that it is effective to use this approach repeatedly to build a strong cyclic plan. The notion involves picking an unhandled reachable state, finding a weak plan from this state in the all-outcomes determinization, and incorporating the plan into the policy. The planner repeats this process until the policy is strong cyclic or it finds a deadend and backtracks to replan.

Similar techniques have been used in online replanning for probabilistic planning problems (e.g., the FF-Replan and FF-Hindsight+ planners [101, 103]). The techniques use both the all-outcomes and single outcome determinization (the latter being informed by the outcome probabilities), and whenever the agent arrives at a state that it has not seen before, it produces a weak plan in a determinization of the problem. Both the approach for strong cyclic planning and online replanning create policies that return the action at the start of a plan that matches the current state. Further, the all-outcomes determinization for finding a weak plan is effective in both online replanning [101] and offline approaches [39].

6.2.3 Illustrative Example

To situate the methods we introduce, we use a problem from the triangle tireworld domain as a running example (cf. Figure 6.1). The objective is to drive from location 11 to 15, however driving from one location to another has the possibility of a tire going flat. If

there is a spare tire at the location of the car (marked by circles in the diagram), then the car can use it to replace a flat. The strategy that maximizes the likelihood of success is to drive directly to location 51 and then over to location 15, as this means a spare tire will always be available in case of a flat.

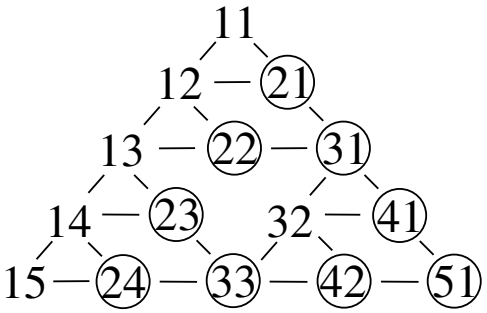


Figure 6.1: Example from the triangle tireworld domain. Circled spots have a spare tire, and the goal is for the car to safely drive from location 11 to 15.

The triangle tireworld domain poses significant difficulty for probabilistic planners [71]. We also found it to be hard for strong cyclic planners due to the attractive nature of driving straight to the goal as well as the vast number of states that the optimal policy can reach. We will refer to the triangle tireworld domain throughout this chapter to demonstrate the impact of our techniques.

6.3 General Approach

To create a strong cyclic plan, we use the all-outcomes determinization to find a weak plan when we encounter a state that our policy does not handle. In the presence of deadends, we replan from scratch rather than backtracking. By exploiting state relevance, we make substantial improvements over previous state-of-the-art methods. This section describes how we build a policy offline, and later outlines how we restrict our offline approach to accomplish online replanning. We first describe our plan representation and the notation surrounding it.

6.3.1 Plan Representation

The solution for a non-deterministic plan is a (potentially partial) mapping from the complete state of the world to an action. However, the naïve representation that uses complete state action pairs can be extremely prohibitive. Many previous state-of-the-art techniques suffer from this limitation (e.g., the previously best-in-class FOND planner,

FIP [39]). To overcome this, we represent the policy using partial states. Formally, our plan representation is defined as follows.

Definition 22 (Partial Policy)**.** A *state-action pair* is a tuple $\langle p, a\rangle$ where p is a partial state and a is an action. A *rule set* RS is a set of state-action pairs and $match_{RS}(s)$ denotes the set of state-action pairs $\langle p, a\rangle \in RS$ such that $s \models p$. If we have a function Φ to select a desired pair from a rule set, the *partial policy* for RS and Φ is the partial function P that maps a partial state s to the desired element in RS: $P(s) = \Phi(match_{RS}(s))$. If P is defined for s, we say that P *handles* s.

Given a rule set and a selection function over the possible state-action pair subsets, we can use the partial policy P to map a given state to an appropriate action: the question remains how to build the rule set and selection function. Our solution solves the all-outcomes determinization for various initial states, creating a weak plan, and our planner then computes a rule set that corresponds to both the actions in the plan and the relevant conditions for the plan to succeed. The selection function we use, Φ_{weak}, chooses the desired state-action pair to be the one that is closest to the goal with ties broken arbitrarily. We associate the action outcome from the weak plan to the corresponding non-deterministic action in the state-action pairs.

As mentioned above, the state in a state-action pair $\langle s, a\rangle$ traditionally corresponds to the complete state s after executing the plan up to just before a. Rather than using the full state, we employ repeated regression from the goal to just before the occurrence of action a in the weak plan. Approaches for execution monitoring use regression to determine precisely which part of the state is relevant in order for the a plan to succeed (e.g., the triangle tables of Fikes *et al.* [34] and the monitoring of plan optimality by Fritz and McIlraith [38]). We use this regressed state as the partial state for the state-action pair and associate the pair with its distance to the goal in the weak plan in which it was found. Formally, if the computed weak plan is $[a_1, \cdots, a, \cdots, a_n]$, then the generated state-action pair for a is $\langle p, a\rangle$, where the condition p is the repeated regression of the goal formula: $p = \mathcal{R}^*(G, [a, \cdots a_n])$.

The actual implementation of our policy stores the rule set in a match tree as described in Section 2.2.4. The set $Pairs$ corresponds to the rule set, and we use the same variable selection function that was introduced in Chapter 5.

6.3.2 Algorithm

We use GENPLANPAIRS(Π, P) to refer to the procedure of creating a weak plan for the all-outcomes determinization and then adding the corresponding state-action pairs

to the policy P. The outcomes chosen by the GenPlanPairs procedure are referred to as the *chosen outcomes*. Algorithm 11 shows our high-level approach to generating a strong cyclic plan. For now, we describe only aspects of Algorithm 11 that do not involve handling deadends, and we describe the details for deadends below.

Algorithm 11: Generate Strong Cyclic Plan

```
   Input: FOND planning task Π = ⟨V, s_0, s_*, A⟩
   Output: Partial policy P
 1 Initialize policy P to the empty policy
 2 while P is the empty policy or P changes do
 3     Open = {s_0};
 4     Seen = {};
 5     while Open ≠ ∅ do
 6         s = Open.pop();
 7         if (s ⊭ s_*) ∧ (s ∉ Seen) then
 8             Seen.add(s);
 9             if P(s) is undefined then
10                 GenPlanPairs(⟨V, s, s_*, A⟩, P);
                   /* P(s) may still be undefined if s is a deadend   */
11             if P(s) is defined then
12                 ⟨p, a⟩ = P(s);
13                 for e ∈ Eff_a do
14                     Open.add(Prog(s, a, e));
15     ProcessDeadends();
16 return P;
```

Algorithm 11 enumerates the reachable states of the policy by considering every outcome of the actions our policy returns (lines 11-14). The algorithm deals with unhandled reachable states by finding a weak plan to achieve the goal and adding the state-action pairs to the policy (lines 9-10).

One difference between Algorithm 11 and FIP, the best-in-class FOND planner prior to PRP, is the outer loop of line 2. We must repeat the process until we find a fixed point because the use of partial states can cause previously handled states to have a new action returned by the policy. If the policy behaves differently, it can lead to a new unhandled open state. We refer to a single execution of lines 2-14 as a *pass*. The check on line 2 succeeds whenever the previous pass has augmented the policy or when a deadend was found. Similar to FIP, GenPlanPairs ceases planning when it finds a state handled by the policy. Stopping the planning process is sound due to the following theorem.

Theorem 12. *If a policy generated by Algorithm 11 handles state s, then following the*

policy along with the chosen action outcomes leads the agent to the goal in a finite number of steps.

Proof. Every state-action pair $\langle p, a\rangle$ in the policy is a part of a weak plan that achieves the goal. Executing the action with the chosen outcome brings the state to a point where the next action (denoted a') in the weak plan is applicable. It may happen that the agent selects the next state-action pair from another weak plan, but this pair must be no further from the goal than a' was (since the policy uses Φ_{weak} for comparison). With this invariant on the state-action pairs, we are guaranteed to reach the goal when following the chosen action outcomes and not encounter an unhandled, deadend, or repeated state. □

The monotonic decrease in the distance-to-goal means that once GenPlanPairs reaches a state that our policy handles, we can always follow the policy and chosen action outcomes to arrive at the goal and generate a weak plan.

6.4 Avoiding Deadends

The previous state-of-the-art planner, FIP, handles deadends by backtracking through the policy updates and searching for a new weak plan that avoids the deadend state. Issues with this approach include failing to recognize the same deadend later on in the search and failing to recognize the core reason for the deadend. In this section, we describe how our approach addresses these two issues to provide a sound and complete search for a strong cyclic policy – i.e., we compute a strong cyclic solution if one exists.

Rather than backtracking, we record the deadends during each pass of Algorithm 11 and avoid them in all subsequent passes. A deadend may arise for two alternative reasons: (1) GenPlanPairs proves no solution exists in the all-outcomes determinization or (2) during the search for a weak plan, the planner discovers a deadend state. We record all instances of both types of deadends and process them in the final step of a pass (ProcessDeadends in Algorithm 11).

6.4.1 Forbidden State-Action Pairs

In classical planning, a deadend state is defined as a state of the world from which the goal cannot be reached. In a non-deterministic setting, we generalize this concept for strong cyclic solutions:

Definition 23 (Deadend)**.** We define a *deadend state* for the SAS$^+$ FOND planning task $\langle \mathcal{V}, s_0, s_*, \mathcal{A}\rangle$ to be a state s such that no strong cyclic plan exists for the planning task

$\langle \mathcal{V}, s, s_*, \mathcal{A} \rangle$. We define a *deadend partial state*, or simply a *deadend*, as a partial state p where for every $s \in \mathcal{S}$ such that $s \models p$, s is a deadend state.

During the search for a strong cyclic solution, we avoid any deadends that exist. Once we have identified a deadend, we leverage regression to identify the situations that can potentially lead to the deadend:

Definition 24 (Forbidden State-Action Pair)**.** We define a *forbidden state-action pair* to be a state-action pair $\langle p, a \rangle$ where p is a partial state, a is an action in our domain, d is a deadend, and the following condition holds:

$$\exists\, e \in \mathit{Eff}_a, \forall\, s \in \mathcal{S}, \quad \text{if} \quad s \models p \quad \text{then} \quad \mathit{Prog}(s, a, e) \models d$$

In other words, a forbidden state-action pair $\langle p, a \rangle$ is a situation in which executing the action a in any state that entails p could lead to a deadend state. A deadend state that cannot reach the goal in the all-outcomes determinization implies that it is a deadend state for the strong cyclic planning problem as well. Given a deadend d that can be regressed through a with effect $e \in \mathit{Eff}_a$, we compute the forbidden state-action pair corresponding to d, a, and e as $\langle \mathit{Regr}(d, a, e), a \rangle$. The forbidden state-action pair signifies that action a should be pruned out in any state matching $\mathit{Regr}(d, a, e)$.

6.4.2 Generalizing Deadends

We have identified two sources of deadend states: (1) states where GenPlanPairs proves no plan exists and (2) states that GenPlanPairs identifies as a deadend during search (i.e., the search heuristic returns a value of infinity). Both sources, however, provide complete states. Note that because these deadends arise in the all-outcomes determinization, we can treat them as classical deadends. In this section we present a method for generalizing deadend states, and in the next section we describe how they are actually used with forbidden state-action pairs during the GenPlanPairs procedure.

To strengthen the deadend reasoning, we *generalize* the deadend states to be a minimal deadend according to the delete relaxation. The delete relaxation of the problem allows a variable to take on more than one value at once [16]. When an action is applied, old variable settings are not removed, and so once an action becomes applicable it is always applicable. It is a simple polynomial operation to see whether or not a plan exists for a state in the delete relaxation of the problem, and we refer to any state where the goal cannot be achieved in the delete relaxation as a *delete relaxed deadend state*. If

a state is a delete relaxed deadend state, then it is necessarily a deadend state for the original problem, but the converse is not necessarily true.

For a partial state p, we can use the delete relaxation to check if it is a deadend for the original problem. The procedure, shown in Algorithm 12, treats the partial state p as the delete relaxed state where every variable not defined in p takes on every possible value at the same time. Because p may be a deadend for the original problem but not a deadend for the delete relaxation, using Algorithm 12 to determine if p is a deadend is sound but incomplete. However, this technique is sufficiently powerful for the deadends we tend to generate.

With a procedure in hand to check if a partial state is a deadend, we use a simple greedy approach to find a minimal deadend given a deadend state. Algorithm 13 is the procedure we use to generalize a deadend state. It will unset variables from the state one after another as long as it remains a delete relaxed deadend (according to IsDeadend).

It may be the case that Algorithm 13 runs through every variable without being able to relax a single one – indeed it may be the case that the entire deadend state s is not a delete relaxed deadend. In such cases, we simply use the full state as the deadend. As an optimization, we do not try to generalize a deadend state that is not also a delete relaxed deadend state.

6.4.3 Processing Deadends

ProcessDeadends begins by generalizing every deadend that was recorded during search. A generalized deadend may be the entire state, but often we generalize a deadend to a much smaller partial state that is the core reason for the deadend.

Next, ProcessDeadends generates a rule set of forbidden state-action pairs by regressing the deadends through every non-deterministic action effect for every action. The rule set captures all of the ways in which a non-deterministic action execution may lead to a deadend. Once we have the rule set for all of the deadends recorded in a pass, we add all of the forbidden state-action pairs to a global rule set used during planning. We require only that a single non-deterministic effect lead to a deadend state for the action to appear in a forbidden state-action pair.

The final step of ProcessDeadends is to reset the policy so that we can start the computation over with the knowledge of forbidden state-action pairs. For GenPlan-Pairs, we modified a forward-search planner to restrict the expansion of nodes in the search frontier to avoid forbidden state-action pairs by filtering the actions applicable at every point in the search. We also modified the heuristic computation to account for

Algorithm 12: Delete Relaxed Deadend Check: IsDeadend(Π, p)

Input: All-outcomes determinization $\Pi = \langle \mathcal{V}, s_0, s_*, \mathcal{A}' \rangle$, and partial state p
Output: *true* if p is detected as a delete-relaxed deadend; *false* otherwise

```
/* S is a set of variable-value pairs to represent the relaxed state. */
S = ∅;
/* Include every variable-value pair for the partial state */
for v ∈ vars(p) do
    S.add( ⟨v, p(v)⟩ );
/* Include every variable-value pair outside of the partial state */
for v ∈ (V − vars(p)) do
    for k ∈ D_v do
        S.add( ⟨v, k⟩ );
/* A is a set of actions we have not applied yet */
A = A';
/* Continually add actions until the goal is reached */
while true do
    /* If the goal is achieved, return false */
    if ∀v ∈ vars(s_*), ⟨v, s_*(v)⟩ ∈ S then
        return false;
    /* Compute the set of applicable actions */
    App = {a | a ∈ A and ∀v ∈ vars(Pre_a), ⟨v, Pre_a(v)⟩ ∈ S};
    /* If there are no applicable actions, return true */
    if |App| = 0 then
        return true;
    /* Apply the applicable actions to the delete relaxed state */
    for a ∈ App do
        /* Note that in the determinization, we only have one effect per action */
        Let Eff_a = {e};
        for v ∈ vars(e) do
            S.add( ⟨v, e(v)⟩ );
    /* Update the set of remaining actions */
    A = A − App;
```

Algorithm 13: Generalization of a Deadend State

Input: All-outcomes determinization $\Pi = \langle \mathcal{V}, s_0, s_*, \mathcal{A}' \rangle$, and deadend state s
Output: Deadend d

```
   /* The deadend d is initialized to the entire state                    */
1  d = s;
   /* Try to relax every variable one after another                      */
2  for v ∈ V do
       /* Relax the variable and check reset it if d is no longer a deadend */
3      old = d(v);
4      d(v) = ⊥;
5      if ! IsDeadend(Π, d) then
6          d(v) = old;
7  return d;
```

the forbidden state-action pairs in order to improve efficiency – this modification does not change the search space fundamentally, as no new deadends are introduced because of the heuristic modifications. We can now establish the soundness and completeness of Algorithm 11.

Theorem 13. *Algorithm 11 computes a strong cyclic plan for a SAS$^+$ FOND planning problem if such a strong cyclic plan exists.*

Proof. For soundness and completeness, we need only look at the final iteration of the outer while-loop. In the final pass, the policy does not change and no new deadend is found. Therefore, the condition on line 9 can only be met if $s = s_0$, in which case no strong cyclic plan exists. Otherwise, we know that the policy handles s_0 and we can follow the policy to eventually achieve the goal for some arrangement of non-deterministic action outcomes (due to Theorem 12). Further, since the condition on line 9 always fails, the policy always returns an action for any state we reach. Since we start with the initial state in the open list (line 4) and enumerate every possible successor for the non-deterministic actions chosen by the policy (lines 13-14), we are guaranteed that the policy represents a strong cyclic plan. Finally, we know that there is a final pass of Algorithm 11 because the policy monotonically adds further state-action pairs to cover unhandled states (line 10) and forbidden state-action pairs to handle discovered deadends. □

6.4.4 Illustrative Example

In the triangle tireworld example, a common deadend found by our planner is for the car to be in a non-goal location with a flat tire and no spare. Consider the case after the action *drive*_11_12 is executed and the tire goes flat, depicted in Figure 6.2 with C representing the car's location and the cross indicating a flat. The car is now in location 12, and without a spare tire has no way of achieving the goal. The full deadend state would contain a setting for every variable in the domain: $has\_flat = True$, $car\_at = 12$, $spare\_in\_11 = False$, $spare\_in\_12 = False$, $spare\_in\_21 = True$, etc.

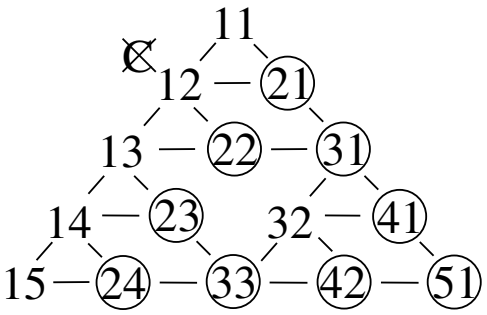


Figure 6.2: Example from the triangle tireworld domain where the car has driven to location 12 causing the tire to go flat.

The generalization of this deadend removes all information pertaining to the location of spare tires in locations other than 12. In this case, the deadend computed by Algorithm 13 contains only the following variable settings:

$$car\_at = 12$$
$$has\_flat = True$$
$$spare\_in\_12 = False$$

The forbidden state-action pairs created for this deadend include any state that has the car one location away without a flat and the action that drives the car to the deadend location. Specifically, these are the four forbidden state-action pairs generated for the generalized deadend:

$$\langle\{car\_at = 11, spare\_in\_12 = False, has\_flat = False\}, drive\_11\_12\rangle$$
$$\langle\{car\_at = 21, spare\_in\_12 = False, has\_flat = False\}, drive\_21\_12\rangle$$
$$\langle\{car\_at = 22, spare\_in\_12 = False, has\_flat = False\}, drive\_22\_12\rangle$$
$$\langle\{car\_at = 13, spare\_in\_12 = False, has\_flat = False\}, drive\_13\_12\rangle$$

The forbidden state-action pairs allow the GENPLANPAIRS procedure to avoid driving to the problematic location 12, either from the initial location 11 or the next position 21.

6.5 Planning Locally

The techniques presented so far are sufficient to produce a strong cyclic plan if one exists. To improve on the efficiency of our approach, in this section we present an extension of a technique introduced in the FIP planner. Later, in Section 6.6, we present a novel technique for determining whether or not the incumbent policy is a strong cyclic plan for a given state.

6.5.1 Approach

One of the significant contributions of the FIP planner was to investigate an unhandled state in the search for a strong cyclic plan in a special way: they sought a local plan to get back to the intended state prior to replanning for the goal (stopping early if the policy handles a state in the search space). We leverage reasoning about partial states to make planning locally more powerful.

Rather than planning for the complete state that we expected our planner to be in, we plan for the expected partial state in our policy that would have matched our expected state. Consider the situation where the following holds for a simulated state s:

1. The policy handles s with $P(s) = \langle p, a\rangle$.
2. The chosen action outcome of a is $e \in \mathit{Eff}_a$.
3. We must consider an unexpected action outcome $e' \in \mathit{Eff}_a \setminus e$ in the simulation.

Planning locally occurs when $P(Prog(s, a, e'))$ is undefined. Instead of searching for a plan from $Prog(s, a, e')$ to $Prog(s, a, e)$, as is the case with FIP, we search for a plan from $Prog(s, a, e')$ to p^* where $P(Prog(s, a, e)) = \langle p^*, a^*\rangle$. Here, p^* is the partial state

that our policy would match if the action outcome occurred as expected. The partial state we plan for contains only the relevant portion of the intended complete state, which is often more succinct and easier to achieve.

When we plan locally we do not record deadends, as a strong cyclic plan that achieves the goal may still exist while a local plan does not. If planning locally fails, the approach is no different than before because the planner records nothing and we subsequently plan for the goal. If planning locally succeeds, it reaches a state that our policy handles and by Theorem 12 we retain soundness and completeness.

The technique can provide a great advantage in certain domains, but we discovered that when we scale the problem size in other domains, planning locally was detrimental to performance. We found that planning locally tends only to provide a benefit when the local plan is extremely short. As such, we limit the search effort when planning locally to expand no more than 100 states in the search space.

6.5.2 Illustrative Example

In the running example, if the agent drives to location 21, which has a spare, and has the chosen outcome of not getting a flat tire, it must handle the case where a flat tire does occur. This example is depicted in Figure 6.3.

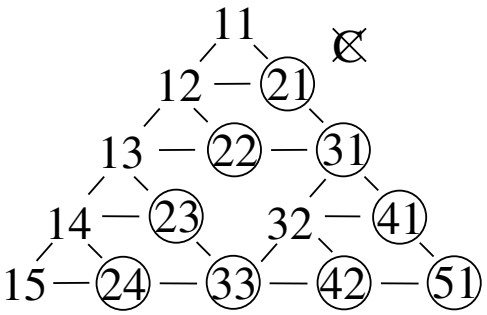


Figure 6.3: Example from the triangle tireworld domain where the car has driven to location 21 causing the tire to go flat.

Without planning locally, the agent would attempt to find an entirely new plan from scratch. If planning locally for a complete state, the agent would attempt to find a plan for the complete state it expected to be in. The problem with this, however, is that the complete state includes the following variable settings among others:

$$car\_at = 21$$
$$has\_flat = False$$
$$spare\_in\_21 = True$$

It is clear to see that the agent can no longer achieve complete expected state, as fixing the tire would necessary change the value of $spare\_in\_21$ to be $False$. However, the expected partial state does not require a spare tire to exist at location 21, but only that we do not have a flat tire:

$$car\_at = 21$$
$$has\_flat = False$$

Planning for the partial local state allows the planner to find a local plan of only one action (replace the tire) instead of finding a new plan for the goal.

6.6 Strong Cyclic Confirmation

Up to this point we exhaustively enumerate every state reachable by a policy to certify that the policy is strong cyclic – Algorithm 11 stops only when the open list of reachable states is empty. We improve on this exhaustive approach by identifying states where our policy acts properly as a strong cyclic plan. Algorithm 11 is modified so that if the policy is known to be a strong cyclic solution for the current simulation state s, lines 13-14 will be skipped. In this section, we describe how state relevance is used to provide a sufficient check for whether or not the policy is a strong cyclic solution.

6.6.1 Approach

The idea behind our test for a strong cyclic policy given a particular state, is to label the state-action pairs in the policy that are necessarily part of a strong cyclic solution.

Definition 25 (Strong Cyclic State-Action Pair)**.** We define a *strong cyclic state-action pair*, with respect to a policy P, to be a state-action pair $\langle p, a\rangle$ such that for any state s where $P(s) = \langle p, a\rangle$, P is a strong cyclic plan for the state s.

If we knew precisely which state-action pairs in our policy were strong cyclic, then we could stop expanding states whenever we arrive at one that corresponds to a strong cyclic state-action pair. The precise condition for a pair to be a strong cyclic, however, is difficult to compute. We instead determine a sufficient condition for a state-action pair to be strong cyclic. Algorithm 14 outlines how we compute pairs that match this condition, which we denote as being *marked*.[2]

Algorithm 14: Mark State-Action Pairs

Input: Planning problem Π and rule set RS
Output: Annotated rule set RS

```
   /* Start with every pair marked                                          */
1  ∀⟨p, a⟩ ∈ RS, ⟨p, a⟩.marked = True;
   /* Iterate until a fixed point is reached                                */
2  while Some pair becomes unmarked do
       /* Iterate through every pair that remains marked and is not a goal  */
3      foreach ⟨p, a⟩ ∈ RS s.t. ⟨p, a⟩.marked ∧ p ⊭ s* do
           /* Iterate over every possible non-deterministic effect          */
4          foreach e ∈ Eff_a do
               /* Unmark if a state could be reached that the policy does not handle */
5              if ∄⟨p', a'⟩ ∈ RS s.t. Prog(p, a, e) ⊨ p' then
6                  ⟨p, a⟩.marked = False;
               /* Unmark if a state could be reached that the policy is not strong cyclic for */
7              else if ∃⟨p', a'⟩ ∈ RS s.t. ⟨p', a'⟩.marked = False ∧ p' ≈ Prog(p, a, e) then
8                  ⟨p, a⟩.marked = False ;
9  return RS;
```

Algorithm 14 works by first assuming that every state-action pair is marked, and then iteratively un-marking those that no longer satisfy the condition to remain marked. The algorithm repeats until it reaches a fixed point, and returns the resulting rule set. Lines 4-8 of the algorithm determine whether or not the state-action pair $\langle p, a\rangle$ should remain marked and there are two conditions that can cause a pair to become unmarked; if either condition holds for some effect $e \in \mathit{Eff}_a$, then we unmark the pair.

The first condition ensures that if the policy returns $\langle p, a\rangle$ for a state s, then there is at least one state-action pair returned by the policy in the state $\mathit{Prog}(s, a, e)$ (line 5). We require this so that if the policy uses $\langle p, a\rangle$, then it handles every possible outcome.

[2] In the published version (cf. Muise *et al.* [76]), there was an error with line 5 of the algorithm that had $p' \models \mathit{Prog}(p, a, e)$ instead of $\mathit{Prog}(p, a, e) \models p'$. The implementation did not suffer from this error.

The second condition ensures that if the policy returns $\langle p, a\rangle$ for state s, every state-action pair returned by the policy in a state reached by executing a is itself marked (line 7). The condition, as presented, may be overly zealous in checking possible state-action pairs, but is nonetheless sound. A better approximation of the applicable pairs from the policy has the potential to leave more pairs marked.

Theorem 14. *If Algorithm 14 leaves a state-action pair $\langle p, a\rangle$ of policy P marked, then for any state s where $s \models p$, P is strong cyclic plan for the state s.*

Proof. Assume for the sake of contradiction that Algorithm 14 completes and there exists a marked state-action pair $\langle p, a\rangle$ that is *not* a strong cyclic state-action pair. Because $\langle p, a\rangle$ remained marked, we know that for any state s if $P(s) = \langle p, a\rangle$, then $\forall e \in \mathit{Eff}_a, P(Prog(s, a, e))$ is defined and further, $P(Prog(s, a, e))$ must be marked. We also know from Theorem 12 that for some $e \in \mathit{Eff}_a$, $P(Prog(s, a, e))$ is a pair closer to the goal than $\langle p, a\rangle$. Inductively, we extend this reasoning until we arrive at goal states which are trivially marked. The collective set of states that we reach serve as a certificate for P to be a strong cyclic plan for state s, and thus violates our assumption that $\langle p, a\rangle$ was not a strong cyclic state-action pair. □

We incorporate marked state-action pairs into Algorithm 11 by adding a condition on line 11 – we only expand the outcomes of an action in a state-action pair if the pair is not marked. The marking of state-action pairs in the rule set for a policy is only valid until we modify the policy. As soon as we introduce a new state-action pair into the policy, we recompute the marking from scratch. Because the policy we compute is usually extremely small, the process of recomputing which pairs should be marked is extremely fast. After line 10 in Algorithm 11, we run Algorithm 14 to compute a valid marking.

To take advantage of the marked state-action pairs, we modify the search conducted by Algorithm 11. Line 6 of this algorithm selects an arbitrary state in the set of open states. FIP explores the states in a breadth first manner, but we do not require this for the algorithm to be sound and complete. We instead explore the states in a depth first manner, investigating open states that are closer to the goal in the weak plans that we find. The exploration works in concert with Algorithm 14 to progressively mark more of the state-action pairs in every iteration.

One final change we make to increase the efficiency of Algorithm 14 is to strengthen the conditions in the state-action pairs of our policy. If the policy returns the state-action pair $\langle p, a\rangle$ for state s with the intended effect $e \in \mathit{Eff}_a$, and we find that a new plan must be computed for the unintended effect $e' \in \mathit{Eff}_a \setminus e$, then we have a newly

created state-action pair $\langle p', a' \rangle$ where $P(Prog(s, a, e')) = \langle p', a' \rangle$. We incorporate the condition for the new plan to succeed into p by replacing $\langle p, a \rangle$ with $\langle p \oplus Regr(p', a, e'), a \rangle$. Since the applicability of a non-deterministic outcome depends only on the precondition of the action, we know that p and $Regr(p', a, e')$ are consistent (i.e., $p \approx Regr(p', a, e')$). If they were not consistent, then we could not have found a plan in the all-outcomes determinization for both of the states $Prog(s, a, e)$ and $Prog(s, a, e')$.

A consequence of strengthening the partial states is that the partial states in the state-action pairs become less general, but the benefits outweigh this drawback. Strengthening allows the condition on line 5 of Algorithm 14 to fail far more frequently, allowing more states to remain marked. The intuition behind this modification is to strengthen the condition in a state-action pair to include the relevant part of the state for all possible outcomes to succeed.

Theorem 15. *Algorithm 11 is a sound and complete procedure for computing a strong cyclic plan if modified to plan locally and if Algorithm 14 is used to stop expanding states.*

Proof. We have already seen that planning locally does not affect the soundness and completeness of Algorithm 11. From Theorem 14 we know that any marked state-action pair is a strong cyclic state-action pair. Looking at the final pass of Algorithm 11 that uses all extensions, we can see that the policy remains unchanged, as does the marking of the state-action pairs in the policy. The search through reachable states would therefore follow the policy's choices, or stop when a marked state-action pair signifies that the policy is a strong cyclic plan for the reached state. Following the exhaustive enumeration of the states and Theorem 14, we can conclude that the final policy returned is either a strong cyclic plan or the empty policy, if no strong cyclic plan exists. Note that planning locally is not essential for soundness and completeness – it only serves to improve the efficiency of the search procedure. □

An interesting side effect of our approach occurs when Algorithm 14 marks the state-action pair, $P(s_0) = \langle p, a \rangle$. When this occurs, p represents the relevant portion of the initial state for our policy to be a strong cyclic solution to the entire planning problem.

6.6.2 Illustrative Example

For the triangle tireworld example, consider the situation where the car is almost at the goal (location 33) as depicted in Figure 6.4. Without the strengthening, we find state-action pairs in the policy where the partial state contains only the location of the car

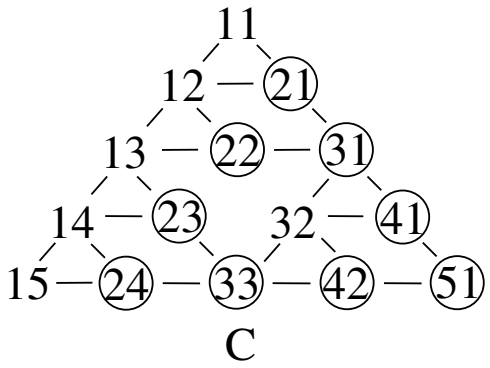


Figure 6.4: Example from the triangle tireworld domain where the car has driven to location 33.

and the fact that the tire is not flat. For our example, this is the pair that is produced without strengthening enabled:

$$p_1 = \langle \{car\_at = 33, has\_flat = False\}, drive\_33\_24 \rangle$$

Strengthening the state-action pair adds the status of spare tires the car may need in the future to the partial state and allows the marking of Algorithm 14 to be much more effective. The strengthened pair corresponding to p_1 is the following:

$$p_2 = \langle \{car\_at = 33, has\_flat = False, spare\_in\_24 = True\}, drive\_33\_24 \rangle$$

To see the advantage of strengthening in the strong cyclic detection, consider the two example pairs p_1 and p_2, along with the additional pairs p_3 and p_4 that complete the plan if the car makes it to location 24 (we assume that p_3 takes precedence over p_4 when both pairs are applicable in a state):

$$p_3 = \langle \{car\_at = 24, has\_flat = False\}, drive\_24\_15 \rangle$$
$$p_4 = \langle \{car\_at = 24, spare\_in\_24 = True\}, fix\_tire\_at\_24 \rangle$$

The pair p_3 will remain marked, as it will always achieve the goal if p_3's condition is satisfied. The pair p_4 will remain marked, as it will always achieve the condition for p_3 if p_4's condition is satisfied. For the pair p_2, consider the result of progressing the partial

state through each non-deterministic outcome of p_2's action:

$$\begin{aligned} &Prog(\{car\_at = 33, has\_flat = False, spare\_in\_24 = True\},\\ &\quad drive\_33\_24,\\ &\quad \{car\_at = 24, \boldsymbol{has\_flat = False}\})\\ =\ &\{car\_at = 24, \boldsymbol{has\_flat = False}, spare\_in\_24 = True\} \end{aligned}$$

$$\begin{aligned} &Prog(\{car\_at = 33, has\_flat = False, spare\_in\_24 = True\},\\ &\quad drive\_33\_24,\\ &\quad \{car\_at = 24, \boldsymbol{has\_flat = True}\})\\ =\ &\{car\_at = 24, \boldsymbol{has\_flat = True}, spare\_in\_24 = True\} \end{aligned}$$

Note that for the first outcome, p_3 will be applicable. Further, for the second outcome, p_4 will be applicable:

$$\{car\_at = 24, has\_flat = False, spare\_in\_24 = True\} \models \{car\_at = 24, has\_flat = False\}$$
$$\{car\_at = 24, has\_flat = True, spare\_in\_24 = True\} \models \{car\_at = 24, spare\_in\_24 = True\}$$

In this case, p_2 will remain marked. If we consider the unstrengthened pair, p_1, we have a different result:

$$\begin{aligned} &Prog(\{car\_at = 33, has\_flat = False\},\\ &\quad drive\_33\_24,\\ &\quad \{car\_at = 24, \boldsymbol{has\_flat = False}\})\\ =\ &\{car\_at = 24, \boldsymbol{has\_flat = False}\} \end{aligned}$$

$$\begin{aligned} &Prog(\{car\_at = 33, has\_flat = False\},\\ &\quad drive\_33\_24,\\ &\quad \{car\_at = 24, \boldsymbol{has\_flat = True}\})\\ =\ &\{car\_at = 24, \boldsymbol{has\_flat = True}\} \end{aligned}$$

While the first outcome adequately entails the applicability of p_3, the second outcome does *not* suffice for p_4 to be applicable. Without knowing if $spare\_in\_24 = True$ holds prior to using p_1, we cannot be sure that the goal is still achievable with the policy. This causes p_1 to be unmarked (due to line 5 of Algorithm 14), and demonstrates the usefulness of strengthening for the strong cyclic confirmation.

Using the depth-first approach of expanding states, the policy increasingly marks the partial states near the goal as the alternatives for fixing the spare tire are explored. For example, in the final passes of Algorithm 11 there will be a pair in the policy for the initial state that includes all of the required spare tires and remains marked:

$$\begin{aligned}\langle\{&car\_at = 11, has\_flat = False, spare\_in\_21 = True,\\ &spare\_in\_31 = True, spare\_in\_41 = True, spare\_in\_51 = True,\\ &spare\_in\_42 = True, spare\_in\_33 = True, spare\_in\_24 = True\},\\ &drive\_11\_21\rangle\end{aligned}$$

6.7 GenPlanPairs

GenPlanPairs is central to our approach, and we have described parts of the procedure throughout the chapter so far. In this section we summarize the previous points and elaborate on the operation of GenPlanPairs.

GenPlanPairs works in two phases: (1) it computes a weak plan for the goal and (2) it computes the condition-action pairs that should be added to the policy. We present an overview of the procedure in Algorithm 15. The most involved step of GenPlanPairs is the WeakPlan procedure: it is an augmented version of a classical planner that takes the incumbent policy and forbidden state-action pairs into account. The WeakPlan procedure uses the all-outcomes determinization of the input problem to compute and return (1) a sequential plan, and (2) a partial state referred to as the "final condition" (explained further below). The plan consists of a sequence of action-outcome pairs that constitutes a classical plan in the all-outcomes determinization, and we refer to the selected outcome as the *chosen outcome*. If no plan is found (i.e., the sequence of action-outcome pairs is empty), we add the initial state to a global list of deadends to be handled later on (cf. line 15 of Algorithm 11). If a plan is returned, the final condition is regressed through every action in the plan to produce new condition-action pairs for the policy P.

In the remainder of this section we detail the key modifications made to the classical planner for the WeakPlan procedure.

Algorithm 15: GenPlanPairs

Input: FOND planning task $\Pi = \langle \mathcal{V}, s_0, s_*, \mathcal{A} \rangle$,
Partial policy P, and
Forbidden state-action pairs F

```
/* Compute a weak plan for the input problem.                    */
(Plan, s'_*) = WeakPlan(⟨V, s_0, s_*, A⟩, P, F);

/* If unsolved, add the initial state as a new deadend.          */
if Plan ≡ [ ] then
    global_deadends.add(s_0);
    return;
else
    Let Plan = [(a_1, e_1), ··· , (a_n, e_n)];

/* Regress the goal condition through the plan.                  */
p = s'_*;
for i = n ··· 1 do
    p = Regr(p, a_i, e_i);

    /* Update the policy.                                        */
    P = P ∪ ⟨p, a_i⟩;
```

Avoiding Forbidden State-Action Pairs

Throughout the search for a classical plan, complete states are continuously expanded to find a plan from the initial state to a state where the goal condition holds. The typical method of expanding a newly reachable state in the search is to consider every applicable action. In the determinization we have an action a_e for every effect e in Eff_a. We can thus define the set of applicable actions as,

$$\mathit{App}(s) = \{a_e \mid s \models \mathit{Pre}_a\}$$

To avoid potentially hazardous parts of the state space, we filter the set of applicable actions with the given set of forbidden state-action pairs, $F = \{\langle p_1, a_1 \rangle, \cdots, \langle p_n, a_n \rangle\}$. This gives us the following updated set of applicable actions:

$$\mathit{App}(s, F) = \{a_e \mid a_e \in \mathit{App}(s) \text{ and } \nexists \langle p, a \rangle \in F, s \models p\}$$

This modification allows the planner to avoid using the forbidden state-action pairs when searching for a solution. If we encounter a state where *every* action is forbidden, then we consider the state to be a deadend and handle it as described below.

Early Stopping Condition

A classical planner generally stops the planning process when it either reaches a state where the goal condition holds or it has exhausted the reachable state space. With the incumbent policy P in hand, we can halt the search early when any state s is reached such that $P(s) \neq \bot$. In such a situation the policy can be used with the chosen outcome for every action returned, and eventually reach a state satisfying the goal.

To take advantage of this property, we check every expanded state to see whether or not the policy can handle it, and stop planning when this is the case. Because we need to construct the new state-action pairs for updating the policy, the WeakPlan procedure must return a "final condition" partial state s'_*. This condition simply corresponds to the partial state that matches the expanded state s in the search procedure: $P(s) = \langle s'_*, a \rangle$. Note that if s satisfies the goal, then s'_* will simply be s_*.

Planning Locally

As described in Section 6.5, we must modify the search procedure to account for planning locally. When WeakPlan is invoked to plan for a condition that is not the goal, we limit the search procedure to a bounded number of state expansions. We use this bound to avoid a large negative impact in performance in domains where planning locally rarely succeeds. We found that using a bound of 100 state expansions provided the best trade-off between dedicating too much time to planning locally and usefulness for domains where planning locally succeeds.

Improved Heuristic

The underlying heuristic used in the classical planner is a modified version of the FF heuristic [51]. Intuitively, the FF heuristic computes a plan for the delete relaxed version of the classical planning problem (i.e., we ignore the delete effects). For the SAS$^+$ encoding that we use, the delete relaxation amounts to a variable accumulating values and never deleting old values.

We modify the heuristic to bias the search towards a plan that avoids forbidden state-action pairs. Previously we have shown how the state expansion avoids forbidden state-action pairs, but here we wish to prefer actions that will lead to areas of the search space with fewer actions that are forbidden. To achieve this behaviour, we restrict the set of actions available for the heuristic computation to those that are *not forbidden* in the current state.

Running the heuristic without the forbidden actions may result in the delete relaxation

becoming unsolvable, even if the state is not a deadend in the determinization. When this happens, we must compute the heuristic again with every action included. This second computation of the heuristic will only occur if the first computation returns a value of infinity (i.e., when no delete relaxed plan exists), and during the second computation we penalize the use of any forbidden action in the delete relaxed solution that makes up the heuristic value.

Because of how we constructed the two heuristic calls, the first will always be larger than the second. When the first heuristic call returns a finite value, the planner will tend to find plans that avoid relying on forbidden state-action pairs. The second heuristic call is necessary, as it guarantees that we correctly identify the delete relaxed deadends we encounter during search.

Recording Deadends

The final modification to the classical planner is the recording of deadends. Whenever the search detects a delete relaxed deadend, the deadend is recorded to later be generalized and turned into forbidden state-action pairs. We additionally record a deadend for any state that has zero applicable actions. These states typically arise when every action applicable in the original domain is forbidden by a state-action pair. Recording the deadends during a search for a weak plan allows PRP to avoid invoking the classical planner later in the search for a strong cyclic policy.

6.8 Strong Plan Recognition

A natural question that arises from our work is whether or not a strong plan can be computed with PRP. Unfortunately, our approach is not amenable to computing a strong plan when cyclic plans exist (i.e., those that visit the same state multiple times). In this section, we discuss some of the issues surrounding the detection of a strong plan.

By definition, any strong plan is also a strong cyclic plan. As a result, PRP will compute a strong plan for any problem that has a strong plan but no cyclic plan. For domains where both strong plans and cyclic plans exist, PRP may or may not find a strong plan during the search. To detect if a strong plan is found, however, we must disable strong cyclic confirmation. When enabled, there is no way to determine if every state reachable by the policy is unique.

We could modify the search procedure to help find strong plans by pruning any applicable action in the search for a weak plan that would lead the planner to a state already reachable by the policy. The resulting algorithm is a sound procedure for computing

strong plans, but it is unfortunately also incomplete; the planner may not find any solution when one exists. The incompleteness arises from the addition of deadend states that are not truly deadends – they are added only because the weak plan could not find a solution that avoids states reachable by the incumbent policy.

The major obstacle for PRP computing a strong plan is the manner in which we exhaustively enumerate the space of potential policies. Rather than backtrack through the decisions made in the weak plan and additions made to the policy, we record the required forbidden state-action pairs and construct a new policy from scratch. While this is an effective technique for producing a strong cyclic solution that avoids deadends, it is unclear as to how the procedure can be modified to compute strong plans instead.

6.9 Offline vs Online

The purpose of our approach is to repeatedly compute a policy until a strong cyclic plan is found. Given sufficient time, the algorithm is sound and complete (cf. Theorem 15), but it is interesting to consider the situation where the planning time is restricted. Our approach to building a policy repeatedly discards the previous policy in the presence of deadends, and then generates a new one that avoids the deadends. While the intermediate policies are not strong cyclic plans, they may still be of use. We compute the quality of an intermediate policy by simulating the domain in a number of trials and observing how often the policy achieves the goal. If the offline approach does not find a strong cyclic plan, either because one does not exist or because the planner has run out of time, it returns the highest quality intermediate policy found.

When the planner is executing actions from the policy in an online phase, it behaves similarly to online replanning approaches for probabilistic planning: if the planner encounters a state that it does not recognize, it computes a weak plan using GenPlanPairs and updates the policy accordingly. When replanning online we augment only the current policy and never reset it. The monotonic nature of our policy construction means that we must ignore forbidden state-action pairs during the execution of GenPlanPairs to deal with a state that the policy cannot handle – even if there is a risk of reaching a deadend, taking *some* action is better than taking no action at all.

By allowing our planner to process the problem offline, we are able to find a policy that systematically avoids deadends in the problem. Not taking deadends into account is one of the major drawbacks of the online replanning approach of FF-Replan [101], and FF-Hindsight addressed this issue by sampling several potential futures before deciding on the action it should take [102, 103]. Our approach is to do this simulation prior to

execution, allowing us to avoid potentially disastrous outcomes and build a more robust policy. By restricting the amount of time given to the offline phase, we can view our approach as an any-time algorithm that represents the full spectrum between an online replanning approach (giving zero time to the offline phase) and a fully offline search (giving the offline phase all of the time it requires).

6.10 Evaluation

We implemented our approach and extensions to FOND planning by augmenting the Fast Downward (FD) planning system [48]. In our planner, PRP, we altered many aspects of FD to make it suitable for non-deterministic planning, including adding a dedicated parser, introducing the translation to a SAS^+ FOND planning task, modifying the inadmissible heuristics to account for forbidden state-action pairs, and implementing optimizations to allow for repeated invocations of the planning procedure.

To ascertain the effectiveness of our approach, we evaluate our planner's efficiency on a range of domains from the FOND benchmark suite and we further evaluate our planner on domains inspired by the probabilistic planning benchmark suites. Whenever possible (and appropriate), we compared PRP to the state-of-the-art FOND planner, FIP. As our planner is not tailored to use the competition software from the probabilistic planning track, we created a special version of our planner, PRP_R, to compare with the techniques employed by FF-Replan. All experiments were conducted on a Linux desktop with a 2.5GHz processor, and we limited the execution of the planners to 30 minutes and 2GB memory.

As in previous work, the run times reported do not include parsing of the domain. For PRP, the parsing time was typically a fraction of a second unless the problem size became prohibitively large. There are many components to PRP, however planning for a weak plan in the determinized domain represents the majority of time our planner takes to find a solution: usually less than 1% of the time is spent in other parts of the overall planning process (i.e., the rest of Algorithm 11).

To evaluate the efficiency of finding a strong cyclic plan we assess the time it takes to compute the plan (in seconds). We also measure the size of the policy using the standard of the number of state-action pairs (smaller being better).[3] When evaluating the online replanning efficiency of our approach, we consider the mean number of actions needed to reach the goal, the mean number of replans needed, and the mean time it takes to reach

[3]While our state-action pairs are using partial states, the underlying implementation is no smaller than storing the complete state.

the goal when executing the plan with our computed policy.

First, we provide a limited description of the previous non-deterministic and probabilistic planners to give a sense of the techniques that have been previously employed. Next, we present the collection of experiments that evaluate the offline planning efficiency for FOND benchmarks, an expository domain, and a collection of probabilistic domains. We additionally include an analysis on the impact some of the added features have on the PRP's performance. We conclude the evaluation with an investigation into the online replanning efficiency of our planner.

6.10.1 Existing Solvers

Symbolic FOND Planners: MBP & Gamer

Along with the initial formalism of weak, strong, and strong cyclic planning, Cimatti *et al.* proposed a series of algorithms to solve non-deterministic planning problems [17]. Their Model Based Planner (MBP) computed strong cyclic and strong plans through careful use of binary decision diagrams that represented a policy for the non-deterministic problem. At every iteration, MBP would grow the decision diagram with the actions that could lead to a state handled by the policy while carefully avoiding those actions that possibly lead to a state outside of the policy. In the case of strong planning, MBP places further restrictions on the policy update to avoid returning to the same state.

The techniques of MBP were extended by Cimatti *et al.* and embedded in the FOND planner, entitled Gamer [58]. The principle behind Gamer is to re-encode the FOND planning problem as a two-player game where the planner plays against the environment by choosing an action while the environment chooses the non-deterministic outcome. The translation primarily serves as a reasonable encoding to enable efficient grounding using existing procedures. Gamer was the only fully observable non-deterministic planner submitted to the 6th International Planning Contest (IPC-2008),[4] and at the time Gamer was the state of the art in non-deterministic planning.

Determinization FOND Planners: NDP & FIP

Following IPC-2008, Kuter *et al.* introduced the planner NDP for FOND problems. NDP took an alternative approach by solving determinized planning problems repeatedly to find a solution [64]. A weak plan is computed with an off-the-shelf classical planner for every reachable state that an incumbent policy does not handle. NDP was a significant

[4] `http://ippc-2008.loria.fr/wiki/index.php/Results.html`

improvement over MBP and Gamer, as it did not rely on exhaustive backward search with a BDD representation. However, NDP suffered from a strong decoupling with the embedded classical planner and lack of focus on the relevant aspects of the state of the world in the policy representation.

Building on the insights of NDP, Fu *et al.* introduced the Fast Incremental Planner (FIP) [39]. There are two key contributions FIP made: (1) the overall search process is coupled with the classical planner so that search for a weak plan ceases whenever it reaches a state the incumbent policy can handle, and (2) when a new weak plan is required, FIP suggests an alternative local goal before planning for the original goal (referred to as planning locally in our work). FIP demonstrated impressive performance on three of the four FOND benchmark domains, but like NDP it suffered from relying on complete states in both the generation and representation of the strong cyclic plan.

Determinization Probabilistic Planners: FF-Replan & FF-Hindsight+

While PRP is not a probabilistic planner, we are still interested in the applicability of the techniques we introduce to typical probabilistic planning problems. One of the original determinize-and-replan approaches for probabilistic planning is the planner FF-Replan [101]. FF-Replan uses an online approach to determinize the problem, and executes actions from the first weak plan found until a non-deterministic outcome occurs that was not expected. At that point, FF-Replan will solve for another weak plan and repeat the process. FF-Replan outperformed every planner from the 2004 and 2006 IPC contests, and additionally performed very well in the 2008 contest.

Following an investigation into the types of domains that appeared in the competition, one prevailing criticism of the FF-Replan approach is its blindness to avoidable deadends [71]. By executing the first weak plan discovered, FF-Replan may end up in a deadend that is entirely avoidable in the domain. To overcome this drawback, various extensions to FF-Replan were introduced in the FF-Hindsight and FF-Hindsight+ planners that attempt to avoid deadends through the careful use of sampling future outcomes [102, 103]. By increasing the number of simulations, the probability of hitting an avoidable deadend is reduced. The drawback here, however, is that a strong cyclic solution is only guaranteed in the limit.

6.10.2 Offline Planning Efficiency

FOND Benchmarks

To measure the efficiency (both time and policy size) of PRP at computing a strong cyclic plan, we first consider the benchmark suite from the FOND track of the 2008 International Planning Competition (IPC). The suite contains four domains: blocksworld (blocks), faults, first responders (first), and forest. Not every problem has a strong cyclic plan. The top of Table 6.1 details the number of problems each planner solved, along with the number of those found not to have a solution.

Domain	**No. of Problems**	**FIP Solved (unsat)**	**PRP Solved (unsat)**
blocks	30	**30** (**0**)	**30** (**0**)
faults	55	**55** (**0**)	**55** (**0**)
first	100	**100** (**25**)	**100** (**25**)
forest	90	20 (11)	**66** (**48**)
blocks-new	50	33 (**0**)	**46** (**0**)
forest-new	90	51 (**0**)	**88** (**0**)
Total	415	289 (36)	**385** (**73**)

Table 6.1: The number of problems for which FIP and PRP either successfully find a strong cyclic plan or prove none exists (the latter is shown in brackets).

PRP has a distinct advantage in the FOND benchmark domains, but both planners are capable of solving the majority of the problems quickly. In the forest domain, where this is not the case, many problems do not have a strong cyclic plan. Both PRP and FIP produce the optimal policy for problems in the faults and first domains – search for a weak plan in these domains goes directly towards the goal, and planning locally always successfully repairs the policy. In the forest domain, the only version of FIP that we had access to falsely identifies 11 problems as having a strong cyclic plan when none exists (these do not appear in the table).

To investigate how the planners scale in the two domains that were not trivially handled, we generated a larger benchmark set for the blocksworld and forest domains. For the forest domain, we modified the problem generator slightly to guarantee that every problem has a strong cyclic plan. For the blocksworld domain, we scaled the number of blocks from 1 to 50 (the highest number of blocks in the original benchmark set is 20). We show the coverage for these domains at the bottom of Table 6.1. For the problems both solved, the top two graphs of Figure 6.5 show the relative time required to compute

and the size of a policy respectively. The final graph of Figure 6.5 shows the coverage for each planner as a function of time. We found that PRP significantly outperformed FIP in these domains, improving both size and run time by an order of magnitude.

Impact of Parameter Settings

To measure the impact our various techniques have on the search for a strong cyclic solution, we ran PRP with various features disabled to test the coverage in the FOND benchmark domains and the triangle tireworld domain mentioned throughout this chapter. Aside from running PRP with the default parameter settings, we additionally used each of the following ablated versions:

- FullLocal: In this version, we restrict PRP to use only full states for planning locally, and do not pre-empt the search for a local plan early (i.e., if no local plan exists, the planner must prove this fact).
- NoSCD: Here, we disable the strong cyclic detection. In this case, the full state space reachable by the policy is explored during the planning procedure.
- NoDeadend: In this version, we disable the deadend generalization and online deadend detection. We do not disable the use of deadends entirely, as it is required for completeness. A deadend in this case, however, will be a complete state.
- FullState: In this final case, we disable the use of partial states. This includes generalizing the deadends found, using partial states for the policy, and planning for a partial local state.

The results for the five approaches (original version and four modifications of PRP) are shown in Table 6.2. For the FOND competition benchmarks, we find very little impact from disabling any single feature – the only change in coverage comes from using the full state in the faults and forest domains. This trend is indicative of a problem structure that has little potential for various weak plans to share sequences of actions. In particular, if using the full state has little impact on the performance, we would not expect strong cyclic detection or generalized deadend techniques to help either.

It is important to note that while the coverage did not drop drastically for the competition domains with an ablated version of PRP, using every technique does not hinder the performance of PRP either. It is reasonable to expect that not every feature will be important for every domain, but we would hope that the fully featured PRP solver is not

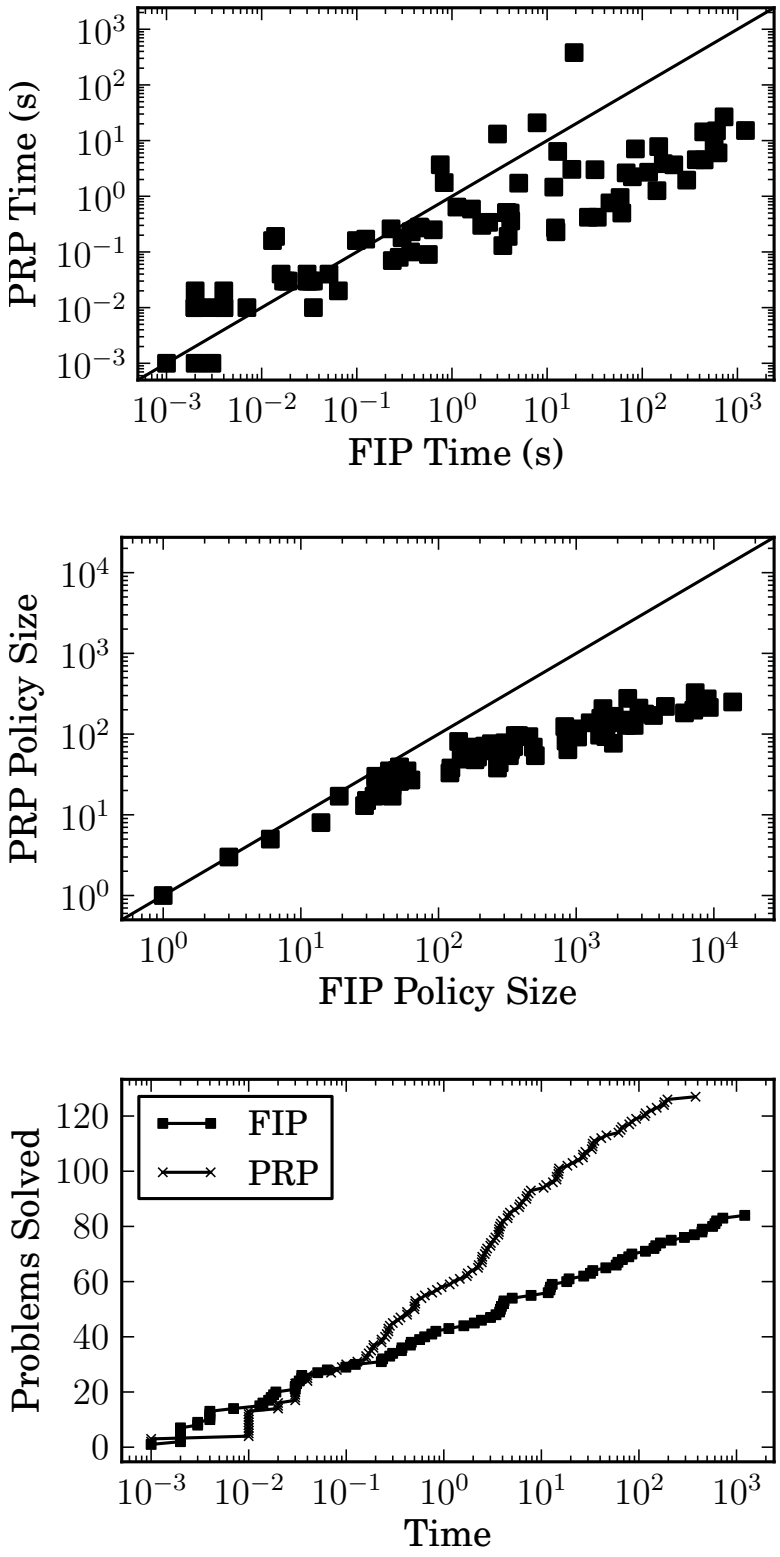


Figure 6.5: Time to find a strong cyclic plan, and size of the plan's policy for PRP and FIP in the newly generated blocksworld and forest benchmark sets. The final figure shows the number of problems solved if given a limited amount of time per problem.

Domain	PRP	FullLocal	NoSCD	NoDeadend	FullState
faults (55)	**55**	**55**	**55**	**55**	46
first (100)	**100**	**100**	**100**	**100**	**100**
blocks-new (50)	**46**	**46**	**46**	**46**	**46**
forest-new (90)	**88**	**88**	**88**	**88**	82
tireworld (40)	**39**	14	5	14	3
Total (335)	**328**	303	294	303	277

Table 6.2: The number of problems for which PRP or a restricted version of PRP either successfully finds a strong cyclic plan or proves none exists.

at a disadvantage in such scenarios. Indeed, this appears to be the case with the FOND benchmark set.

Finally, considering the new challenging domain, triangle tireworld, we observe a striking drop in coverage if any one feature is disabled. The motivating example text throughout this chapter provides same intuition as to why each feature is important. Most critically, in the triangle tireworld domain there are exponentially many paths to the goal and the optimal policy can reach exponentially many distinct states. Having a compact representation of the policy is essential to PRP's performance in this domain.

Impact of Relevance

To demonstrate an extreme example where state relevance plays a vital role in computing a policy, we introduce the concept of irrelevant action outcomes.

Definition 26 (Irrelevance)**.** We define an *irrelevant variable* as a variable $v \in \mathcal{V}$ such that $s_*(v) = \bot$ and $\forall a \in \mathcal{A}, Pre_a(v) = \bot$. That is, v is never relevant for the applicability of an action or the goal. We define an *irrelevant action outcome* e_i to be an outcome of action a that differs from another outcome of a by only irrelevant variables: $\exists e \in Eff_a \setminus e_i, \forall v \in \mathcal{V}, e_i(v) = e(v)$ or $e_i(v) = \bot$ or v is irrelevant.

While this notion of irrelevance is extreme, it serves as artificial approximation of irrelevance that occurs in many of the existing domains. Typically, irrelevance surfaces when an action can achieve multiple fluents, but only a subset of them are required for the particular weak plan that is computed by the GenPlanPairs algorithm. The presence of irrelevant action outcomes causes a planner that uses the full state to spend unnecessary time generating a strong cyclic plan. In the extreme, the performance of a planner using the full state is exponentially worse.

Theorem 16. *In domains with irrelevant action outcomes, our approach computes a policy exponentially smaller in the number of irrelevant action outcomes than the smallest policy computed by existing methods that use the entire state.*

Proof. Consider the case in which every action randomly changes a set of irrelevant variables. The number of irrelevant action outcomes is exponential in the number of irrelevant variables and a strong cyclic plan must be able to handle every one of them. Using only the relevant portion of the state means that our approach only needs to find a plan for the relevant action outcomes. The states reached by irrelevant action outcomes match a partial state in our policy, while approaches using complete states must replan. □

To investigate the impact of state relevance, we modified the original blocksworld benchmark domain to have a varying number of irrelevant variables that every action non-deterministically flips to true or false (full domain details can be found in Appendix B.5). Due to limitations of FIP on the number of non-deterministic effects, we compare a version of our planner that uses complete state for everything (PRP_{Full}) to a version that uses partial states for the policy it constructs (PRP). The degree of irrelevance, k, was varied between 1 and 5 variables creating up to a factor of $2^5 = 32$ times the number of non-deterministic outcomes for each action. Out of 150 problems, using the full state solves only 85 problems while using the partial state solves 130 problems – the remaining 20 problems have too many ground actions to be parsed. Figure 6.6 shows (1) the time comparison for all values of k (from 1 to 5), (2) the size comparison for all values of k, and (3) the number of problems solved for each approach as a function of time. We find a clear exponential separation between the two approaches, and for higher k the difference in time to find a solution and size of the policy is up to several orders of magnitude.

Probabilistic Domains

In probabilistic planning, the objective is to compute a policy that achieves the goal with high probability [42]. Rather than having simply non-deterministic outcomes, the actions in a probabilistic planning problem have probabilistic outcomes – every outcome has a probability associated with it and the sum of all outcome probabilities for an action is 1.0. As an approximation for domains with probabilistic action outcomes, we ignore the probabilities and treat the problem as a FOND planning task. Note that if a strong cyclic solution exists for the FOND approximation, then the resulting policy will be optimal for the probabilistic planning problem. It is with this observation that we investigate benchmark domains from the probabilistic planning that yield strong cyclic solutions.

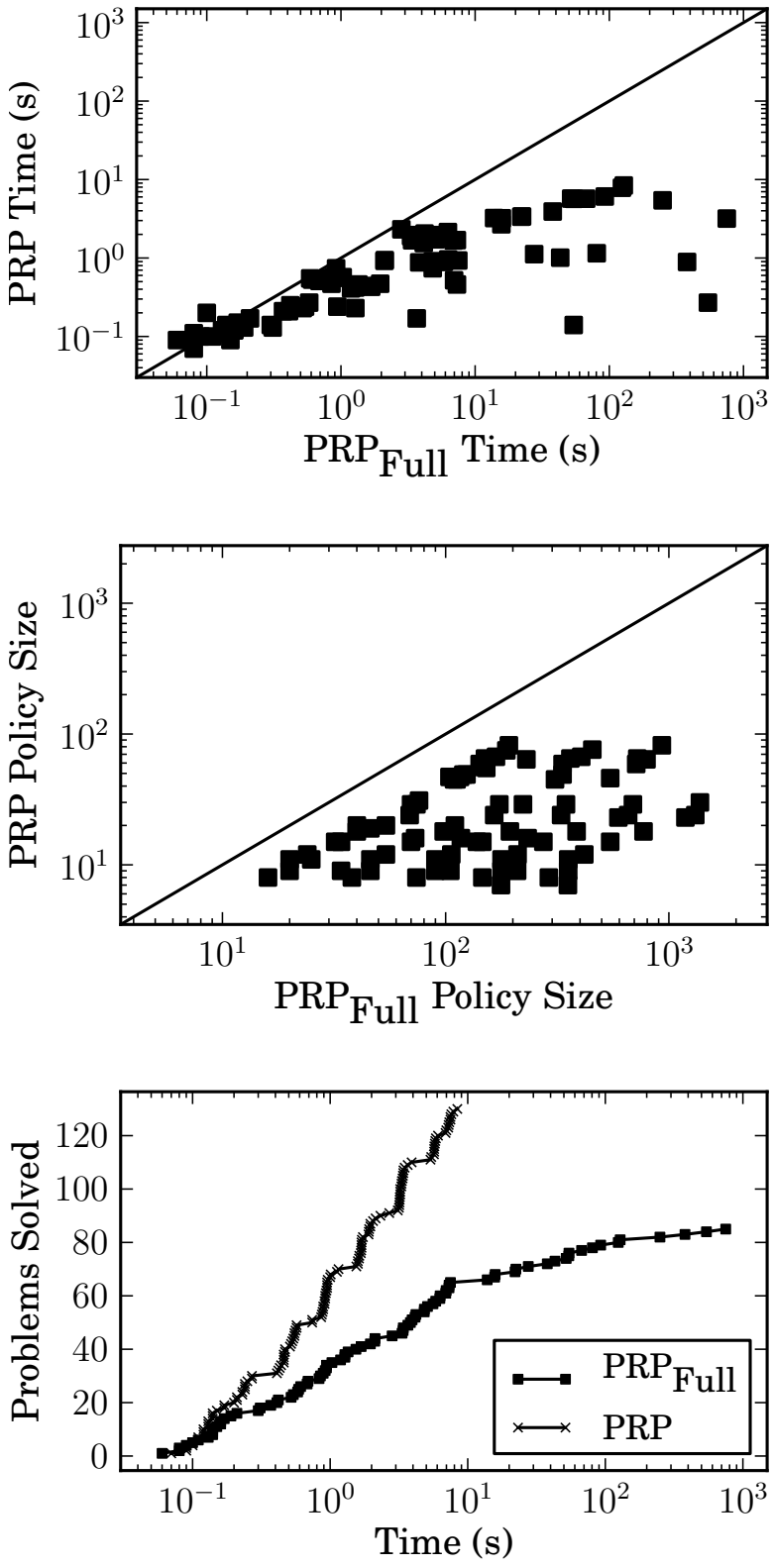


Figure 6.6: Time to find a strong cyclic plan, and size of the plan's policy for PRP and PRP_{Full} in the blocksworld domain with irrelevant fluents added. The final figure shows the number of problems solved if given a limited amount of time per problem.

Domain (# Probs)	Success Rate (%)		Total Time (sec.)	
	FF-H+	PRP	FF-H+	PRP
blocksworld-2 (15)	74.4	**100**	900	**8.4**
elevators (15)	64.9	**100**	1620	**1.7**
zenotravel (15)	68.9	**100**	1620	**98.7**
climber (1)	**100**	**100**	-	0
river (1)	**66.7**	**66.7**	-	0
bus-fare (1)	**100**	**100**	-	0
tire-1 (1)	**100**	**100**	-	0
tire-17 (1)	**100**	**100**	-	14.2
tire-39 (1)	-	**100**	-	1691.9

Table 6.3: Percentage of successful runs out of 30 per problem for PRP and FF-Hindsight+ in a range of domains.

We investigate three domains from the IPC-2006 probabilistic planning track: elevators, blocksworld-2, and zenotravel. We also considered the four "probabilistically interesting" domains introduced by Little and Thiébaux [71]: climber, river, bus-fare, and triangle-tireworld (tire). In Table 6.3 we show the percentage of successful runs (out of 30 per problem) and the total time spent computing a policy for all of the problems in a domain. For comparison, we include the results for FF-Hindsight+ (FF-H+) reported in [103] ('-' indicates unreported results). Note that while the experimental machine we used is slightly faster, the FF-H+ results serve as a rough comparison.

While our experimental setup is not identical to the probabilistic IPC track, the efficiency of PRP in finding a strong cyclic plan in these domains is striking. For the three domains from the probabilistic planning track, we see an improvement of several orders of magnitude to find a solution that is far more robust – PRP succeeds with 100% probability while FF-Hindsight+ fails up to 35% of the time. Most notable is the ability of PRP to solve large triangle tireworld problems. Previously, FF-Hindsight+ claimed the record for this domain by solving tire-17, but through the advances introduced in this chapter, PRP is capable of solving tire-39 in under 30 minutes.

6.10.3 Online Replanning Efficiency

To measure the efficiency of our approach for computing plans online and replanning in the presence of unexpected states, we gave PRP zero time to process the domain before it started execution. We compared ourselves to PRP_{R}: a version of our planner that behaves like FF-Replan, using complete states rather than partial ones and also having

Domain	Avg. Actions		Avg. Replans		Avg. Time	
	PRP_R	PRP	PRP_R	PRP	PRP_R	PRP
bw-new	77.4	**75.8**	**5.3**	**5.3**	3.2	**2.6**
fr-new	**60.4**	65.7	3.9	**3.5**	0.6	**0.2**
bw-irr-2	17593.3	**73.6**	67.2	**3.0**	3.4	**1.2**

Table 6.4: Mean number of actions, replans, and time (in seconds) for PRP and PRP_R to reach the goal.

zero time to process the domain before execution. Similar to the offline case, we see little improvement in the standard benchmark set, as the problem instances are all fairly trivial for online replanning (with the notable exception of the forest domain where deadends are common and online replanning fails consistently). We also found this to be the case for the probabilistic planning problems that yield strong cyclic plans.

We instead present results on the generated blocksworld (bw-new) and forest (fr-new) benchmarks described earlier, as well as the blocksworld domain with two irrelevant fluents added (bw-irr-2). Domains with further irrelevant fluents caused PRP_R not to complete the majority of problems generated. Table 6.4 presents the average number of actions, states requiring replanning, and time to achieve the goal.

We observe an increase in performance on the newly generated benchmarks, but most notably a striking improvement in the number of actions required in a domain with irrelevant fluents. The variance for the number of replans and time to achieve the goal is fairly low for PRP_R, but the number of actions required is extremely high in a handful of problems. The large increase is due to the planner's poor choices when replanning for the goal and ending up in a cycle of states that it has low probability of escaping. There is always a non-zero chance of escaping, which is why both PRP and PRP_R have perfect coverage in the tested domains.

6.11 Discussion

In this chapter, we presented a suite of techniques that exploit state relevance to achieve state-of-the-art performance in non-deterministic planning. Our planner, PRP, is capable of finding a strong cyclic plan for a FOND problem several orders of magnitude faster than the previous state of the art, and generates a policy representation that can be exponentially smaller. We demonstrated PRP's proficiency in domains from both FOND and probabilistic planning contests. The key contributions that lead to the success of PRP include a compact representation of the policy that leverages relevant portions of

the state, a robust approach for handling deadends in a non-deterministic domain, and a novel method to compute sufficient conditions for a policy to be a strong cyclic plan.

In this section, we review some related work and conclude with a discussion of our approach and where we may take it in the future.

6.11.1 Related Work

There have been a variety of methods explored for solving non-deterministic and probabilistic planning problems. A recent trend is to leverage the great strides classical planning has made on improving the scalability of modern planners, and this is the approach we follow for our work. Here, we review some the related approaches and point out the connections various components of PRP have with the literature.

Approaches to compute strong cyclic plans are directly related to our work. FIP is the most recent and effective solver for FOND problems prior to ours [39]. Mattmüller *et al.* propose an LAO* search based on pattern database heuristics to solve SAS$^+$ FOND planning tasks [72] and older approaches use either general game playing or symbolic model checking [58, 17]. Algorithm 14 is also related to the strong extension phase presented in Cimatti *et al.* [17]. The key difference is that we search only through the state-action pairs in our policy when marking; rather than searching all possible pairs in the domain. Additionally, our approach may return a strong cyclic policy that is not a strong policy, even if a strong policy exists.

We have already seen the probabilistic approaches that are related to our work in Section 6.10.1. The approach we take to limit the processing of a domain offline is similar to the notion of "just-in-case scheduling", where the idea is to plan for contingencies in a scheduling problem before the execution phase begins [31]. In a similar fashion, our framework allows for a preset amount of computational effort to be spent systematically computing a policy that will avoid potential deadends.

Our use of relevance to construct a policy is inspired by approaches for execution monitoring (e.g., Fritz and McIlraith [38] and our work from Chapter 4). These approaches use regression to determine relevant portions of the state to build a policy for online execution. Similar approaches have been proposed for probabilistic planning to compute a so-called *basis function* for the existence of a weak plan that has a non-zero probability of reaching the goal [90, 61]. The deadend generalization we employ is also related to computation of deadends in Kolobov *et al.* [62]; the key difference is how our initial candidate is chosen. The condition strengthening we use for strong cyclic detection is related to the regression of a formula through a non-deterministic action to determine

relevance [87]. Both aim to compute the relevant portion of the state for a formula to hold after the execution of a non-deterministic action.

Finally, there are two recent publications that leverage our work in non-deterministic planning for other paradigms in the field of automated planning. First, Ramirez *et al.* use the PRP planner for behaviour composition by recasting the problem of synchronizing multiple behaviours into a single controller [85]. The non-determinism in their setting stems from the partial controllability of the various behaviours that are to be combined. The second approach that relies on PRP is due to Patrizi *et al.* [81]. In this work, temporally extended goals (specified in linear temporal logic) are compiled into a Büchi automata, which is subsequently encoded as part of the planning domain. The original domain theory contains actions with non-deterministic effects, and a solution to the final domain corresponds to a policy that operates in a non-deterministic setting while satisfying the temporally extended goal infinitely often. In both works, the use of PRP was demonstrated to be a key contributor to the notable improvement over the previous state of the art in terms of both performance and scalability of the proposed approach.

6.11.2 Conclusion

While we employ a replanning strategy in an offline phase, PRP is guaranteed to compute a strong cyclic plan if one exists. This is not typically the case with the online planning techniques that employ replanning to heuristically avoid potential deadends in the problem. Our framework for computing the policy additionally allows us to run PRP as an any-time algorithm that returns the best quality policy found after a given amount of time. In this work, we measure quality as the observed likelihood of success over a number of simulations with the policy. The improvements we make to computing a policy for planning in the presence of non-deterministic action outcomes represent a significant step forward in planner capabilities for this type of problem. As a highly efficient solver for FOND planning problems, PRP opens the door to solving new paradigms that can be recast as a FOND planning problem (e.g., [85, 81]).

In the future, we hope to improve the reasoning in domains where action outcome probabilities are known by following an approach similar to scheduling techniques that focus on highly probable state outcomes [9]. In a similar vein, we would like to consider the extension of our work to other planning paradigms that have a relation to non-deterministic planning including conditional or contingent planning where sensing plays a role. In these domains, sensing can be viewed as a form of non-deterministic action where the outcomes correspond to the sensing result [1]. We also plan to develop better

techniques for reasoning with deadends in non-deterministic domains that combine the knowledge of related forbidden state-action pairs. Finally, we intend to investigate the reason behind the success of planning locally: if the best plan leading to the goal goes through the expected state, then planning for the goal should be just as effective as planning locally. Continually attempting to get back to the main plan may result in a policy that is less robust or has a higher expected plan length.

Chapter 7

Concluding Remarks

7.1 Summary

In this dissertation, we introduced a range of techniques to improve how an agent can deliberate and act in a dynamic environment. Under the umbrella of automated planning, we focused on the areas of plan generation, execution, and representation (including both *what* we represent as an agent's plan and *how* it is represented). All three are key aspects of automated planning, and one must employ efficient methods for them in order to realize an effective planning system. We aimed to improve two facets of how an agent interacts with the world: *flexibility* and *robustness*. The flexibility of an executing agent refers to the amount of discretion the agent has when performing actions. Robustness, on the other hand, refers to the agent's ability to continue executing towards a goal without the need to replan. The two concepts are often intertwined – improving the flexibility of an agent can lead to an increase in robustness.

We improved the flexibility and/or robustness of various aspects of planning through exploiting a unifying concept: *relevance*. In particular, we introduced and leveraged three notions of relevance. *Ordering relevance* refers to a minimal subset of all possible ordering constraints in a plan that are sufficient for the plan to be valid. *State relevance* refers to a minimal subset of the state that is sufficient for a property of interest to hold (e.g., for the state to be a deadend). Finally, *temporal relevance* refers to the minimal subset of an agent's execution history that is sufficient for a property of interest to hold (e.g., testing the continued satisfaction of a set of temporal constraints). All three forms of relevance contribute to improving the flexibility and robustness of some part of the planning process.

Improving the flexibility and robustness of plan generation, execution, and representation has a significant impact on the planning process. The contributions we present in

this dissertation enable agents based on planning technology to avoid potentially costly replanning and operate in a dynamic or uncertain environment. In order to best improve flexibility and robustness, the contributions we introduce span all three areas of the planning process.

The thesis of this dissertation is that logical techniques for determining what is relevant to efficiently generate, execute, and represent a plan can improve the overall efficiency, flexibility and robustness of the planning process. In the remainder of this section, we detail our contributions and discuss potential avenues of further research.

7.2 Contributions

We have made a number of contributions to improve the flexibility and robustness of automated planning techniques used by agents acting in a dynamic environment. Most generally, we have demonstrated how different notions of relevance can be exploited to improve on existing techniques for plan generation, execution, and representation. In this section, we summarize the individual contributions made in each of the four main chapters in this dissertation.

Optimally Relaxing Partial-Order Plans (Chapter 3)

We considered the problem of optimally relaxing the ordering constraints of a partial-order plan in Chapter 3. To address this problem, we presented a novel partial-weighted MaxSAT encoding where solutions correspond to maximally flexible partial-order plans. Moving beyond considering only the ordering constraints of a plan, we introduced the notion of a *minimum cost least commitment partial-order plan*, characterized by having as small total action cost as possible and subsequently as few ordering constraints as possible. The encodings we introduced were shown to compute the various notions of optimally relaxed partial-order plans depending on the set of auxiliary constraints that are included with the model of the problem. We further empirically demonstrated that an existing heuristic approach for relaxing plans is extremely adept at computing a particular type of optimal relaxation, despite the algorithm's poor theoretical guarantee. Finally, we demonstrated the feasibility of our approach empirically and compared our methods to previous related work. Our approach is able to compute plans that maximize the inherent flexibility, and can do so in 16% more of the problems than the previous approach for computing a minimum reordering.

Robustly Executing Partial-Order Plans (Chapter 4)

In Chapter 4 we presented a method for generalizing a partial-order plan (POP) for robust and efficient execution. We introduced a theoretical characterization of POP viability, as opposed to the traditional notion of POP validity, which identifies the situations under which some linearization of the POP can achieve the goal. This characterization provides a compelling alternative for the definition of regression through a POP. We introduced a procedure that uses our notion of POP viability to implicitly enumerate every plan fragment of the POP and compute the sufficient condition for the fragment to achieve the goal. We then presented a method for compactly representing these conditions as a policy, and identified the characteristics of a POP that maximize flexibility. The resulting system is far more efficient and robust than previous methods, and we demonstrated this through an extensive empirical evaluation. In our work, we measured efficiency by the time required for an action to be chosen during execution, and we measured robustness by analytically computing the total number of states handled by the generalized plan. Our methods produce generalized plans that are up to 2.5 times more robust than the plans produced by previous approaches. Further, we found the efficiency improvement over previous methods to be up to a factor of 17, indicating a significant benefit for using the policy in a real-time setting.

Robustly Executing Temporally Constrained Plans (Chapter 5)

In Chapter 5 we introduced a compelling framework and semantics for describing temporal constraints over the execution of a plan rather than the construction of a plan. We proposed a set of three temporal constraints between a pair of actions and two temporal constraints between an action and the state of the world. These constraints serve to disambiguate the interpretation of a temporal relationship between a pair of events when one event can occur multiple times. The overall result is an execution scheme that combines techniques from plan execution and schedule dispatching to effectively execute a temporally constrained plan. The plan execution component of Chapter 5 builds on the insights from the work in Chapter 4, as the input plan to be executed is a temporally constrained partial-order plan. To accomplish the temporal reasoning required, we appealed to a variety of schedule dispatching techniques. Through careful analysis in an offline phase, we were able to restrict the temporal reasoning during execution to a small relevant subset of a potentially large execution history. The resulting system is able to dynamically

reason about both logical and temporal requirements for a plan fragment to achieve the goal. We both provided an implementation of the proposed system, and proved the desired properties of the framework. The work in Chapter 5 serves as a unifying approach to both the execution of a POP and the dispatching of a temporal network.

Efficient Non-deterministic Planning (Chapter 6)

We focused on the paradigm of fully observable non-deterministic (FOND) planning in Chapter 6. In this chapter, we developed a state-of-the-art planner for solving FOND planning problems that substantially improves on the previous state of the art. We presented a suite of principled methods to leverage the relevant part of the state of the world for FOND planning, including (1) targeted replanning when the incumbent solution is insufficient, (2) compact and robust representation of the solution, (3) efficient dead-end detection and avoidance, and (4) powerful stopping conditions for the algorithm's simulation loop. Our system is able to compute robust solutions exponentially faster than previous methods, and the resulting policies are also exponentially smaller. The contributions represent a significant improvement in the scalability of non-deterministic planning and introduce the possibility for better probabilistic planning in domains that are difficult due to avoidable dead-ends. Since its introduction, our planner has been leveraged to produce state-of-the-art performance in two other areas of automated planning (cf. the work of Ramirez *et al.* [85] and Patrizi *et al.* [81]).

7.3 Future Work

We have already described potential avenues of future research specific to each of main contributions in their respective chapters. Here, we focus on research directions that take a broader view, and potentially combine or extend multiple facets of the work presented in this dissertation.

Generalizing Richer Inputs

Both Chapters 4 and 5 consider taking a partial-order plan (potentially augmented with temporal constraints) as input to a generalization procedure. The structure of the approaches follows a very similar strategy: enumerate the set of possible partial plan fragments and compute the conditions under which the fragments can achieve the goal.

Extending this strategy to other formalisms of plan representation or domain control knowledge would allow for a far richer set of inputs to the plan generalization procedure.

The first extension we might consider is to generalize a Hierarchical Task Network (HTN) [32]. An HTN is a specification that describes how complex tasks can be decomposed through various methods into sub-tasks. Typically, the aim of an HTN problem is to find a decomposition of the tasks (i.e., an appropriate method for each complex task) until all of the remaining tasks are primitive (i.e., classical planning actions). Every decomposition in the space of valid decompositions corresponds to a sequence of actions. By systematically enumerating the tasks at the end of an HTN, the space of possible sequential plans can be enumerated to produce a list of condition-action pairs, just as we do in the methods of Chapter 4.

Another possible input representation suitable for generalizing is a GOLOG program [69]. GOLOG is an agent programming language that has a number of powerful constructs to control the possible behaviours of an agent. Parts of a GOLOG program can be left unspecified, and it is up to the interpreter to find a valid sequence of actions that realizes the program. The range of possible GOLOG programs is defined through a set of possible compositions of smaller GOLOG program components. This set can be leveraged to compute the range of possible action sequences in reverse order, similar to how we construct the linearization of a POP by starting with just the goal action. Following this intuition, a GOLOG program can be processed to iteratively yield the set of action sequences that realize some fragment of the GOLOG program, and regression can be leveraged to compute the condition required for the fragment to achieve the goal. While regression has previously been used to assess if a GOLOG program can achieve a goal formula [86], the novelty of our method stems from the generalized plan we could produce using regression.

The prevailing theme for all of these representations (i.e., POPs, HTNs, and GOLOG programs) is that they all implicitly embody a family of plans. It is this family of plans that we systematically consider as potential recipes to achieve the goal.

Alternate Plan Composition

One limitation to the approach we introduce for generalizing a plan in Chapters 4 and 5 is that only a single plan is considered for generalization. The policy produced in Chapter 6 does not suffer from this limitation, as we iteratively improve the policy offline with new ways of achieving the goal. The generality of the condition-action pairs, however, lends itself nicely to composing the generalization of entirely different plans into one policy.

When we consider the interpretation of a condition-action pair, we find that it provides

three key things: (1) the condition under which the pair should be considered (i.e., using it will lead to the goal eventually), (2) the action that should be executed next, and (3) a measure of quality so as to choose the "best" pair. If we were to merge the list of condition-action pairs from multiple plans, our overall approach would seamlessly switch between the plans, selecting the most appropriate pair for the given situation.

The one aspect that must be maintained between the different plans is the measure of quality for a pair. As long as we have a unique metric for measuring the quality of every condition-action pair, then the generalization procedure can remain agnostic as to where the pairs come from – be it multiple plans to achieve the same goal or even multiple plans represented in different forms (e.g., the combination of a GOLOG program and a POP). Further, if part of the condition in a condition-action pair includes the final goal that must be achieved, we can use the same process to generalize a set of plans with heterogeneous goals.

Finally, if the aim is to compute a policy that is robust as possible, the techniques of Chapter 3 can be modified to produce plans that are as "different" as possible from another existing plan. By incorporating a notion of diversity into the optimization criteria, we can produce a set of plans with complementary strengths when it comes time to execute them.

Pro-active Policy Repair

The abstract search procedure presented in Chapter 6 aims to find states reachable but not handled by the policy. In a sense, the policy is repeatedly patched so that it becomes more robust over time. We can apply the same intuition to the work of Chapters 4 and 5 to pro-actively repair the vulnerable aspects of the computed policy – those situations that the policy will not be able to handle.

There are two natural approaches for identifying the vulnerable parts of a policy. First, we can simulate the execution if we have some model of the dynamics in the domain – this is the approach taken in Chapter 6 where we know the set of possible non-deterministic outcomes. The second approach places no assumption on having a true model of the world. Instead, we focus on the policy representation itself – this is the approach taken in Chapters 4 and 5.

Because our policy representation is either an OADD (cf. Chapter 4) or a Match Tree (cf. Chapters 5 and 6), we have an effective means of syntactically analyzing the representation to identify the situations unhandled by the policy. For example, in Chapter 4 we use the special $\perp$ symbol to represent situations where the policy does not return an action. The conditions under which our policy fails will be given as a partial state.

Combined with an efficient approach for deadend detection (e.g., the methods introduced in Section 6.4), we can check if the vulnerable policy condition is either unreachable from the initial state or a generalized deadend for the original goal. If neither case holds, we can plan from some initial state not handled by the policy which matches the vulnerable condition. This new plan, if found, could be used to strengthen the policy so that part of the vulnerable condition is now handled.

A number of interesting questions are raised when considering how to make a policy more robust. The two we identify here are how to locate the vulnerable aspects of the policy and how to perform targeted repair of the policy. Other issues worth considering for future work include learning the unknown dynamics of the world online (e.g., similar to the work of Zhuo *et al.* [107]) and modifying the planning process to produce plans that are easier to generalize.

Appendix A

Relaxer Counterexample

The Relaxer Algorithm presented in Chapter 3 deorders an input plan, but as pointed out by Bäckström [4], the resulting POP may not be a minimal deordering. The counterexample provided by Bäckström, however, incorrectly states that resulting POP is not minimally deordered [4, Figure 14]. Here, we present a new counterexample that supports the claim that the Relaxer Algorithm may not produce a minimum deordering.

Both the domain theory and problem specification are shown in Figure A.1. The input plan is the sequence of actions $[a_1, a_2, a_3]$. Because the Relaxer Algorithm seeks out the earliest achiever for every precondition, the algorithm results in a deordering of the plan that has two ordering constraints: $(a_1 \prec a_3)$ and $(a_2 \prec a_3)$. The problem with the deordering is that a_1 is chosen as the achiever for the fluent p, when in fact a_2 can be used as the achiever for both p and q (note that a_2 is already required for fluent q).

The weakness of the Relaxer Algorithm is that it uses the earliest achiever. This weakness surfaces when an action later in the plan can be used as an achiever, and that action is already ordered appropriately. Using this insight, there may be a modification of the Relaxer Algorithm that finds achievers already ordered appropriately, as opposed to finding the earliest achiever.

```
(define (domain counterexample)
  (:requirements :strips)
  (:predicates  (p) (q) (g1) (g2) (g3) )

(:action a1
  :parameters()
  :precondition ()
  :effect (and (g1) (p)))

(:action a2
  :parameters()
  :precondition ()
  :effect (and (g2) (p) (q)))

(:action a3
  :parameters()
  :precondition (and (p) (q))
  :effect (and (g3))))

(define (problem counterexample-problem)
   (:domain counterexample)
   (:init )
   (:goal (and (g1) (g2) (g3) )))
```

Figure A.1: Counterexample Domain and Problem Description

Appendix B

Domains

In this appendix, we give the precise domain description and example executions for the domains that we introduce throughout the dissertation.

B.1 Parallel Domain

The parallel planning domain for parameter k requires k independent actions in order to achieve the goal. There is no interference between the effects of the actions, and Figure B.1 shows an example plan with $k = 3$.

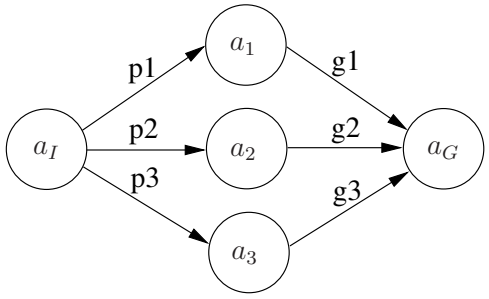


Figure B.1: Example of the Parallel domain with $k = 3$.

B.1.1 Action Theory

The domain theory for the parallel domain is shown in Figure B.2 and the problem description is shown in Figure B.3.

```
(define (domain parallel)
  (:requirements :strips)
  (:predicates  (p1) (p2) (p3) (g1) (g2) (g3) )

(:action a1
  :parameters()
  :precondition (and (p1))
  :effect (and (g1)))

(:action a2
  :parameters()
  :precondition (and (p2))
  :effect (and (g2)))

(:action a3
  :parameters()
  :precondition (and (p3))
  :effect (and (g3)))

)
```

Figure B.2: Parallel Domain Description

```
(define (problem parallel-3)
   (:domain parallel)
   (:init (p1) (p2) (p3))
   (:goal (and (g1) (g2) (g3) )))
```

Figure B.3: Parallel Problem Description

B.1.2 Example Executions

Any ordering of the actions in a solution to the parallel domain is a valid plan. For $k = 3$, the following linearizations include every valid plan:

- a_1, a_2, a_3
- a_1, a_3, a_2
- a_2, a_1, a_3
- a_2, a_3, a_1
- a_3, a_1, a_2
- a_3, a_2, a_1

B.2 Dependent Domain

The dependent planning domain for parameter k requires k pairs of actions such that if they are ordered correctly, fewer fluents from the initial state are required. Figure B.4 shows a fragment of an example plan.

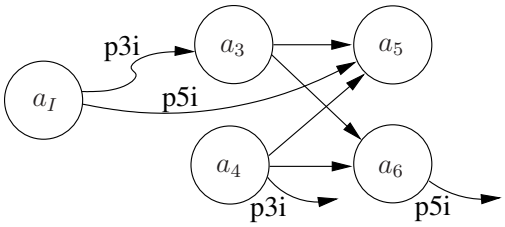


Figure B.4: Excerpt from the POP of a Dependent domain problem. An edge with no endpoint signifies that the action has that fluent as an effect. We only label edges of interest.

B.2.1 Action Theory

The domain theory for the dependent domain is shown in Figure B.5 and the problem description is shown in Figure B.6. We include an explicit a_I action so that the optimal linearization succeeds with certainty.

```
(define (domain dependent)
  (:requirements :strips)
  (:predicates  (p11) (p12) (p21) (p22) (p31) (p32) (p41)
                (p42) (p51) (p52) (p61) (p62) (p1i) (p3i) )


(:action aI
  :parameters()
  :effect (and (p11) (p12) (p21) (p22) ))

(:action a1
  :parameters()
  :precondition (and (p11) (p12) (p1i))
  :effect (and (p31) (p41)))

(:action a2
  :parameters()
  :precondition (and (p21) (p22))
  :effect (and (p32) (p42) (p1i)))

(:action a3
  :parameters()
  :precondition (and (p31) (p32) (p3i))
  :effect (and (p51) (p61)))

(:action a4
  :parameters()
  :precondition (and (p41) (p42))
  :effect (and (p52) (p62) (p3i)))


)
```

Figure B.5: Dependent Domain

```
(define (problem dependent-2)
   (:domain dependent)
   (:init (p1i) (p3i) )
   (:goal (and (p51) (p52) )))
```

Figure B.6: Dependent Problem

B.2.2 Example Executions

In the dependent domain, the pairs of actions can be ordered arbitrarily, but the pairs must follow a prescribed order. For $k = 2$, the following linearizations include every valid plan:

- a_I, a_1, a_2, a_3, a_4
- a_I, a_1, a_2, a_4, a_3
- a_I, a_2, a_1, a_3, a_4
- a_I, a_2, a_1, a_4, a_3

B.3 Tail Domain

The tail planning domain for parameter k requires a sequence of k actions followed by a specially designated pair of unordered actions. Figure B.7 shows an example plan for the tail domain with $k = 3$.

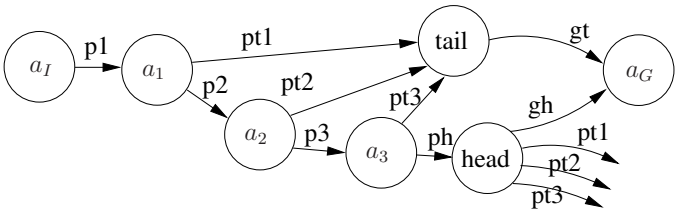


Figure B.7: Example of the Tail domain with $k = 3$. An edge with no endpoint indicates an unused action effect.

B.3.1 Action Theory

For $k = 3$, the domain theory for the tail domain is shown in Figure B.8 and the problem description is shown in Figure B.9.

B.3.2 Example Executions

In the tail domain, the only unordered actions are the *head* and *tail* actions. This leaves the two possible linearizations for the tail domain with $k = 3$:

- $a_1, a_2, a_3, head, tail$
- $a_1, a_2, a_3, tail, head$

```
(define (domain tail)
  (:requirements :strips)
  (:predicates  (p1) (pt1) (p2) (pt2) (p3) (pt3) (gt) (ph) (gh) )

(:action a1
  :parameters()
  :precondition (and (p1))
  :effect (and (p2) (pt1)))

(:action a2
  :parameters()
  :precondition (and (p2))
  :effect (and (p3) (pt2)))

(:action a3
  :parameters()
  :precondition (and (p3))
  :effect (and (ph) (pt3)))
)

(:action head
  :parameters()
  :precondition (and (ph))
  :effect (and (gh) (pt1) (pt2) (pt3)))

(:action tail
  :parameters()
  :precondition (and (pt1) (pt2) (pt3))
  :effect (and (gt)))

)
```

Figure B.8: Tail Domain

```
(define (problem tail-3)
   (:domain tail)
   (:init (p1))
   (:goal (and (gh))))
```

Figure B.9: Tail Problem

B.4 Cognitive Assistant Domain

The Cognitive Assistant domain is a representative example of the daily activities someone may need to perform. The domain was constructed to challenge the temporal reasoning presented in Chapter 5, and so the focus is on the temporal constraints found within the domain. Unlike the other domains discussed here, the Cognitive Assistant domain has a plan already specified. We first present the domain theory that is used for the actions in the plan as well as the temporal constraints among them. We then describe how the dynamic world is simulated and include an example execution that is found through a simple dispatching technique.

B.4.1 Domain Theory

The following fluents make up the state of the world in the Cognitive Assistant domain:

- *at_home*: The user is at home.
- *at_school*: The user is at school.
- *at_mall*: The user is at the mall.
- *awake*: The user is awake.
- *sleeping*: The user is sleeping.
- *hungry*: The user is hungry.
- $full$: The user is not hungry.
- *laundry_clean*: The laundry is finished.
- *kitchen_clean*: The kitchen is clean.
- *have_groceries*: The user has groceries.
- *book_read*: The user has read their book.
- *movie_watched*: A movie has been watched.
- *kids_home*: The kids are home.
- *kids_school*: The kids are at school.
- *have_meal*: A meal is ready to be served.

With the above fluents, the following actions appear in the constructed TPOP for the Cognitive Assistant domain (recall that an action a is a tuple made up of the three sets $\langle PRE(a), ADD(a), DEL(a)\rangle$):

$$
\begin{aligned}
&drive\_kids\_home: \\
&\quad \langle \{awake, at\_school, kids\_school\}, \\
&\quad \{at\_home, kids\_home, hungry\}, \\
&\quad \{at\_school, kids\_school, full\}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&drive\_kids\_school: \\
&\quad \langle \{awake, at\_home, kids\_home\}, \\
&\quad \{at\_school, kids\_school\}, \\
&\quad \{at\_home, kids\_home\}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&drive\_home: \\
&\quad \langle \{awake\}, \\
&\quad \{at\_home\}, \\
&\quad \{at\_school, at\_mall\}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&drive\_school: \\
&\quad \langle \{awake\}, \\
&\quad \{at\_school\}, \\
&\quad \{at\_home, at\_mall\}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&drive\_mall: \\
&\quad \langle \{awake\}, \\
&\quad \{at\_mall\}, \\
&\quad \{at\_home, at\_school\}\rangle
\end{aligned}
$$

$do\_laundry$:

$\langle \{at\_home, awake\},$
$\{laundry\_clean\},$
$\{\}\rangle$

$cook\_meal$:

$\langle \{at\_home, awake, kitchen\_clean, have\_groceries\},$
$\{have\_meal\},$
$\{kitchen\_clean\}\rangle$

$serve\_meal$:

$\langle \{at\_home, awake, have\_meal\},$
$\{full\},$
$\{hungry, have\_meal\}\rangle$

$serve\_breakfast$:

$\langle \{at\_home, awake, kids\_home\},$
$\{full\},$
$\{kitchen\_clean\}\rangle$

$clean\_kitchen$:

$\langle \{at\_home, awake\},$
$\{kitchen\_clean\},$
$\{\}\rangle$

$get\_groceries$:

$\langle \{at\_mall, awake\},$
$\{have\_groceries\},$
$\{\}\rangle$

$read\_book$:

$\langle \{at\_home, awake, kids\_school\},$
$\{book\_read\},$
$\{\}\rangle$

$watch\_movie$:

$\langle \{at\_mall, awake\},$
$\{movie\_watched\},$
$\{\}\rangle$

$wakeup$:

$\langle \{at\_home, sleeping\},$
$\{awake\},$
$\{sleeping\}\rangle$

$goto\_sleep$:

$\langle \{at\_home, awake\},$
$\{sleeping\},$
$\{awake\}\rangle$

$check\_at\_mall$:

$\langle \{at\_mall\},$
$\{\},$
$\{\}\rangle$

$check\_at\_home$:

$\langle \{at\_home\},$
$\{\},$
$\{\}\rangle$

$$check\_full:$$
$$\langle\{full\},$$
$$\{\},$$
$$\{\}\rangle$$

$$check\_clean\_kitchen:$$
$$\langle\{kitchen\_clean\},$$
$$\{\},$$
$$\{\}\rangle$$

The initial state of the problem is the following (recall that all fluents not mentioned are presumed to be false):

$$\{at\_home, sleeping, hungry, kitchen\_clean, kids\_home\}$$

Finally, the goal of the Cognitive Assistant domain is to achieve the following state:

$$\{at\_home, sleeping, full, laundry\_clean, have\_groceries,$$
$$book\_read, movie\_watched, kids\_home\}$$

B.4.2 Temporal Constraints

There are a number of ordering constraints restricting the possible linearizations of the plan, and they are depicted in Figure B.10. The temporal constraints arise from two sources: (1) those specified between actions and (2) those that correspond to durative actions. The list of temporal constraints of the former variety is as follows:

- **(latest-before** *drive_kids_school serve_breakfast* 10 ∞**)**
- **(latest-before** *watch_movie check_at_mall* 30 ∞**)**
- **(latest-before** *cook_meal check_clean_kitchen* 10 ∞**)**
- **(earliest-after** *do_laundry goal* 240 1020**)**
- **(earliest-after** *do_laundry check_at_home* ϵ 10**)**
- **(earliest-after** *serve_meal goal* 300 ∞**)**

- **(latest-before** *wakeup init* 360 480**)**
- **(latest-before** *goto_sleep wakeup* ϵ 960**)**
- **(earliest-after** *wakeup check_full* ϵ 60**)**
- **(latest-before** *drive_kids_school init* 480 510**)**
- **(latest-before** *goal init* 1440 1440**)**
- **(latest-before** *drive_kids_home init* 900 930**)**

The durative actions are split into a start and end time point, and their durations are restricted through the following constraints:

- $30 \leq duration(drive\_kids\_school) \leq 30$
- $30 \leq duration(drive\_kids\_home) \leq 30$
- $20 \leq duration(drive\_mall) \leq 20$
- $20 \leq duration(drive\_home) \leq 20$
- $20 \leq duration(drive\_school) \leq 20$
- $30 \leq duration(do\_laundry) \leq 60$
- $90 \leq duration(watch\_movie) \leq 120$
- $10 \leq duration(cook\_meal) \leq 30$
- $20 \leq duration(clean\_kitchen) \leq 20$
- $60 \leq duration(get\_groceries) \leq 60$
- $30 \leq duration(read\_book) \leq 180$

To use a more realistic setting, an appropriate time point was used for the constraints that occur between a durative action and some other action. Whenever one of the *drive* actions is used in a temporal constraint, the starting time point is used. For all other durative actions, the ending time point is used when it appears in a temporal constraint. For example, the constraint **(latest-before** *drive_kids_school serve_breakfast* 10 ∞**)** is translated to the constraint **(latest-before** $drive\_kids\_school_{start}$ *serve_breakfast* 10 ∞**)** while the constraint **(earliest-after** *do_laundry goal* 240 1020**)** is translated to the constraint **(earliest-after** $do\_laundry_{end}$ *goal* 240 1020**)**.

B.4.3 Environment Variability

To model a dynamic world, certain fluents in the state change according to a variability parameter α. In this section, we describe precisely how α is used to determine if a fluent should change. The values we present were constructed by testing the range of possible settings and observing which extremes were most appropriate (i.e., allowing a solution to exist in at least some of the simulations).

The simulation loop of the Cognitive Assistant domain consists of two phases: (1) the agent selects the next action to execute and the state is progressed, and (2) the state of the world is perturbed according to α. For any given fluent f we define two probabilities respectively corresponding to when f becomes true or false: $p_{add}(f)$ and $p_{del}(f)$. During the second phase, fluent f is (1) made true with probability $p_{add}(f)$, (2) made false with probability $(1-p_{add}(f))\cdot p_{del}(f)$, or (3) left as is with probability $(1-p_{add}(f))\cdot(1-p_{del}(f))$. We compute $p_{add}(f)$ and $p_{del}(f)$ as follows:

$$p_{add}(f) = \alpha \cdot max_{add}(f)$$
$$p_{del}(f) = \alpha \cdot max_{del}(f)$$

Finally, for all fluents f, we define the values of $max_{add}(f)$ and $max_{del}(f)$ to be 0 with the following exceptions:[1]

- $max_{add}(laundry\_clean) = 0.16$
- $max_{del}(laundry\_clean) = 0.5$
- $max_{add}(full) = 0.16$
- $max_{del}(full) = 0.4$
- $max_{add}(book\_read) = 0.16$
- $max_{add}(movie\_watched) = 0.16$
- $max_{add}(have\_groceries) = 0.16$
- $max_{del}(kitchen\_clean) = 0.5$

[1]When $full$ is added (respectively deleted), *hungry* is deleted (respectively added).

B.4.4 Example Execution

In Chapter 5, we considered a dynamic world where an action may occur multiple times. For a static environment, however, every action in the constructed plan occurs once. The following timed execution is an example where the environment is static.

```
[0.000000] init
[420.000000] wakeup
[450.000000] serve_breakfast
[465.000005] check_full
[495.000000] startA_drive_kids_school
[525.000000] endA_drive_kids_school
[607.499975] startA_drive_mall
[627.499975] endA_drive_mall
[668.749963] check_at_mall
[689.374962] startA_get_groceries
[749.374962] endA_get_groceries
[759.687466] startA_watch_movie
[854.843713] endA_watch_movie
[857.421847] startA_drive_home
[877.421847] endA_drive_home
[878.710914] startA_do_laundry
[879.355452] startA_read_book
[909.355442] endA_do_laundry
[909.677711] check_at_home
[909.838851] endA_read_book
[909.919426] startA_drive_school
[929.919426] endA_drive_school
[929.959718] startA_drive_kids_home
[959.959718] endA_drive_kids_home
[1039.979854] startA_clean_kitchen
[1059.979854] endA_clean_kitchen
[1089.989927] check_clean_kitchen
[1114.994959] startA_cook_meal
[1132.497475] endA_cook_meal
[1136.248743] serve_meal
[1258.124377] goto_sleep
```

B.5 Irrelevant Blocksworld

The irrelevant blocksworld domain is a modification of the standard non-deterministic blocksworld domain that introduces k switches that non-deterministically flip after every action. The switches are irrelevant because they are not used as a precondition or goal.

B.5.1 Action Theory

For $k = 3$, the a fragment of the domain theory for the irrelevant blocksworld domain is shown in Figure B.11. We only include a subset of the actions to demonstrate the modification that includes flipping the switches non-deterministically. For completeness, we include a small example of a problem description in Figure B.12.

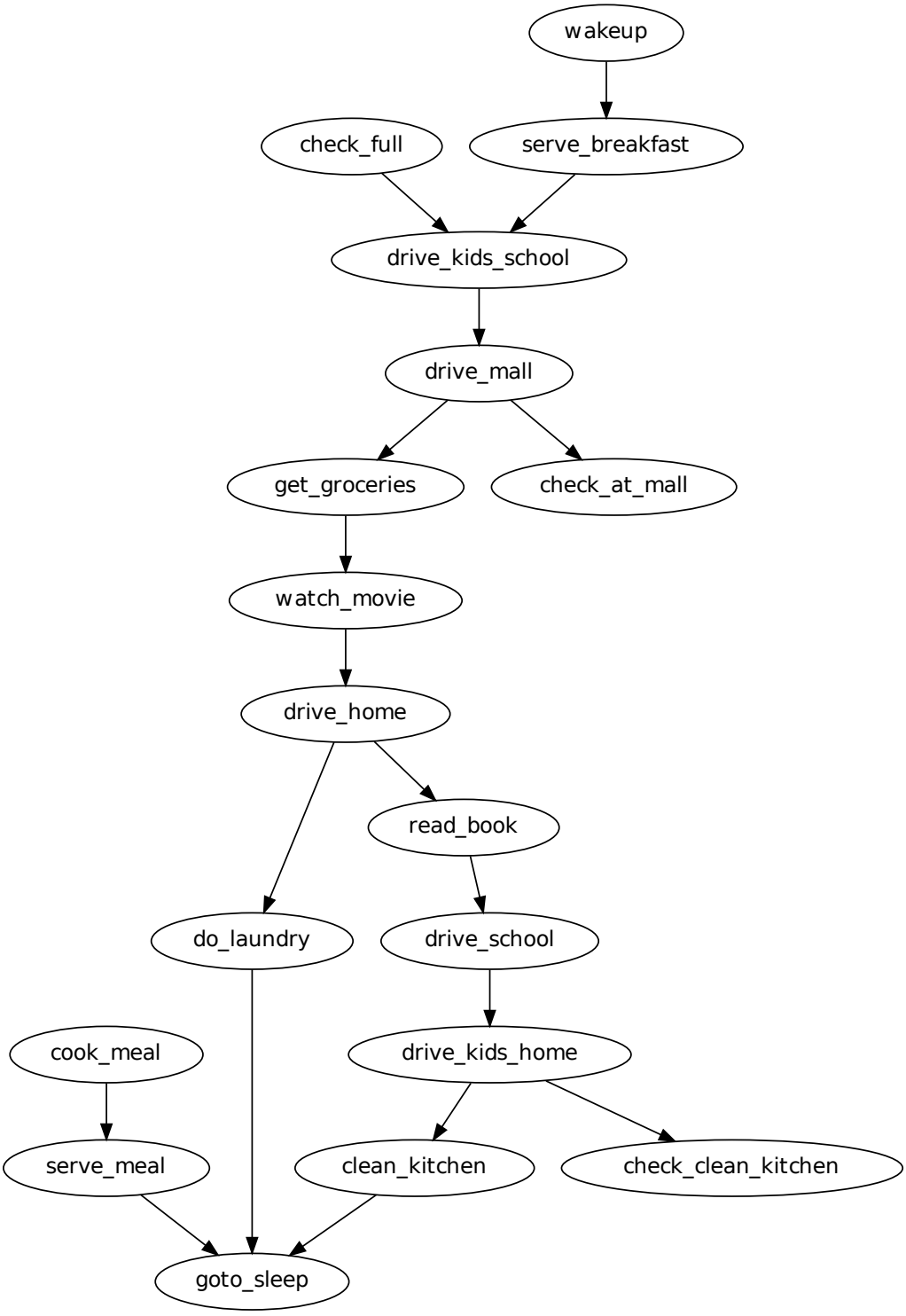


Figure B.10: Ordering constraints for the Cognitive Assistant domain TPOP.

```
(define (domain blocks-domain)

  (:requirements :non-deterministic :negative-preconditions :equality :typing)

  (:types block)

  (:predicates (holding-one) (holding-two) (holding ?b - block) (emptyhand)
               (on-table ?b - block) (on ?b1 ?b2 - block) (clear ?b - block)
               (switch1) (switch2) (switch3))

  (:action pick-up
    :parameters (?b1 ?b2 - block)
    :precondition (and (not (= ?b1 ?b2)) (emptyhand) (clear ?b1) (on ?b1 ?b2))
    :effect (and (clear ?b2) (not (on ?b1 ?b2))
             (oneof (and (holding-one) (holding ?b1) (not (emptyhand))
                         (not (clear ?b1)))
                    (on-table ?b1))
             (oneof (switch1) (not (switch1)))
             (oneof (switch2) (not (switch2)))
             (oneof (switch3) (not (switch3))))
  )

  (:action put-down
    :parameters (?b - block)
    :precondition (and (holding-one) (holding ?b))
    :effect (and (on-table ?b) (emptyhand) (clear ?b)
                 (not (holding-one)) (not (holding ?b))
             (oneof (switch1) (not (switch1)))
             (oneof (switch2) (not (switch2)))
             (oneof (switch3) (not (switch3))))
  )

   ...

)
```

Figure B.11: Irrelevant Blocksworld Domain

```
(define (problem bw_5_1)
  (:domain blocks-domain)
  (:objects b1 b2 b3 b4 b5 - block)
  (:init (emptyhand) (on b1 b3) (on b2 b1) (on-table b3)
         (on-table b4) (on b5 b4) (clear b2) (clear b5))
  (:goal (and (emptyhand) (on b1 b2) (on b2 b5) (on-table b3)
              (on-table b4) (on-table b5) (clear b1)
              (clear b3) (clear b4)))
)
```

Figure B.12: Irrelevant Blocksworld Problem

Bibliography

[1] Alexandre Albore, Hector Palacios, and Hector Geffner. A Translation-based Approach to Contingent Planning. In *21st International Joint Conference on Artificial Intelligence*, pages 1623–1628, 2009.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] John S. Anderson and Arthur M. Farley. Plan abstraction based on operator generalization. In *7th International Conference on Artificial Intelligence*, pages 100–104, 1988.

[4] Christer Bäckström. Computational aspects of reordering plans. *Journal of Artificial Intelligence Research*, 9(1):99–137, 1998.

[5] R. Iris Bahar, Erica A. Frohm, Charles M Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebric decision diagrams and their applications. *Formal Methods in System Design*, 10(2):171–206, 1997.

[6] Jorge A. Baier and Sheila A. McIlraith. Planning with temporally extended goals using heuristic search. In *16th International Conference on Automated Planning and Scheduling*, pages 342–345, 2006.

[7] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4), 2007.

[8] David A. Bell and Hui Wang. A formalism for relevance and its application in feature subset selection. *Machine Learning*, 41(2):175–195, 2000.

[9] Julien Bidot. *A general framework integrating techniques for scheduling under uncertainty*. PhD thesis, École Nationale d'Ingénieurs de Tarbes, France, 2005.

[10] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of satisfiability, frontiers in artificial intelligence and applications*. IOS Press, 2009.

[11] Marcus Bjäreland. *Model-based execution monitoring*. PhD thesis, Linköping University, 2001.

[12] Blai Bonet and Hector Geffner. Causal Belief Decomposition for Planning with Sensing: Completeness Results and Practical Approximation. In *23rd International Joint Conference On Artificial Intelligence*, pages 2275–2281, 2013.

[13] Craig Boutilier and Ronen I. Brafman. Partial-order planning with concurrent interacting actions. *Journal of Artificial Intelligence Research*, 14(1):105–136, 2001.

[14] Craig Boutilier, Richard Dearden, and Moisés Goldszmidt. Exploiting structure in policy construction. In *14th International Joint Conference on Artificial Intelligence*, pages 1104–1113, 1995.

[15] Daniel Bryce and Olivier Buffet. 6th International Planning Competition: Uncertainty Track. In *Proceedings of the International Planning Competition*, 2008.

[16] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 1994.

[17] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso. Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, 147(1):35–84, 2003.

[18] Andrew Coles, Amanda Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *20th International Conference on Automated Planning and Scheduling*, pages 42–49, 2010.

[19] Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. Planning with problems requiring temporal coordination. In *23rd Conference on Artificial Intelligence*, pages 892–897, 2008.

[20] Patrick R. Conrad and Brian C. Williams. Drake: an efficient executive for temporal plans with choice. *Journal of Artificial Intelligence Research*, 42(1):607–659, 2011.

[21] Olivier Coudert. On solving covering problems. In *33rd Annual Design Automation Conference*, pages 197–202, 1996.

[22] William Cushing, Subbarao Kambhampati, Mausam, and Daniel S. Weld. When is temporal planning really temporal. In *20th International Joint Conference on Artificial Intelligence*, pages 1852–1859, 2007.

[23] Marco Daniele, Paolo Traverso, and Moshe Y. Vardi. Strong cyclic planning revisited. In *5th European Conference on Planning*, pages 35–48. Springer, 2000.

[24] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[25] Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up maxsat solving. In *19th International Conference on Principles and Practice of Constraint Programming*, pages 247–262, 2013.

[26] Sammy Davis-Mendelow, Jorge A. Baier, and Sheila A. Mcilraith. Making reasonable assumptions to plan with incomplete information. In *ICAPS Workshop on Heuristic Search for Domain Independent Planning*, 2012.

[27] Giuseppe De Giacomo, Raymond Reiter, and Mikhail Soutchanski. Execution Monitoring of High-Level Robot Programs. In *6th International Conference on Principles of Knowledge Representation and Reasoning*, pages 453–465, 1998.

[28] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[29] Minh B. Do and Subbarao Kambhampati. Improving the temporal flexibility of position constrained metric temporal plans. In *AIPS Workshop on Planning in Temporal Domains*, 2003.

[30] Patrick Doherty, Jonas Kvarnström, and Fredrik Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, 2009.

[31] Mark Drummond, John Bresina, and Keith Swanson. Just-in-case scheduling. In *12th National Conference on Artificial Intelligence*, pages 1098–1104, 1994.

[32] Kutluhan Erol, James Hendler, and Dana S Nau. HTN planning: Complexity and expressivity. In *12th National Conference on Artificial Intelligence*, pages 1123–1128, 1994.

[33] Patrick Eyerich and Malte Helmert. Stronger Abstraction Heuristics Through Perimeter Search. In *23rd International Conference on Automated Planning and Scheduling*, pages 303–307, 2013.

[34] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial intelligence*, 3(1):251–288, 1972.

[35] Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20(1):61–124, 2003.

[36] Maria Fox, Derek Long, and Keith Halsey. An investigation into the expressive power of PDDL2.1. In *16th European Conference of Artificial Intelligence*, pages 338–342, 2004.

[37] Christian Fritz. *Monitoring the generation and execution of optimal plans*. PhD thesis, University of Toronto, 2009.

[38] Christian Fritz and Sheila A. McIlraith. Monitoring plan optimality during execution. In *17th International Conference on Automated Planning and Scheduling*, pages 144–151, 2007.

[39] Jicheng Fu, Vincent Ng, Farokh B. Bastani, and I-Ling Yen. Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems. In *22nd International Joint Conference On Artificial Intelligence*, pages 1949–1954, 2011.

[40] Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668, 2009.

[41] Alfonso Gerevini, Anna Perini, and Francesco Ricci. Incremental algorithms for managing temporal constraints. In *8th IEEE International Conference on Tools with Artificial Intelligence*, pages 360–363, 1996.

[42] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers, 2004.

[43] John R. Graham, Keith S. Decker, and Michael Mersic. DECAF - A Flexible Multi Agent System Architecture. *Autonomous Agents and Multi-Agent Systems*, 7(1):7–27, 2001.

[44] Russ Greiner and Devika Subramanian. AAAI Fall Symposium Series on Relevance. 1994.

[45] Charles Gretton and Sylvie Thiébaux. Exploiting first-order regression in inductive policy selection. In *20th Conference in Uncertainty in Artificial Intelligence*, pages 217–225, 2004.

[46] Inc. Gurobi Optimization. Gurobi Optimizer Reference Manual, 2013.

[47] Patrik Haslum, Malte Helmert, and A Jonsson. Safe, strong and tractable relevance analysis for planning. In *23rd International Conference on Automated Planning and Scheduling*, pages 317–321, 2013.

[48] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1):191–246, 2006.

[49] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

[50] Sarah Hickmott and Sebastian Sardina. Optimality properties of planning via Petri net unfolding: A formal analysis. In Alfonso Gerevini and Adele Howe, editors, *19th International Conference on Automated Planning and Scheduling*, pages 170–177, Thessaloniki, Greece, September 2009. AAAI Press.

[51] J Hoffmann and B Nebel. The FF planning system: fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14(1):253–302, 2001.

[52] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Integrated task and motion planning in belief space. *International Journal of Robotic Research*, 32(9-10):1194–1227, 2013.

[53] Subbarao Kambhampati and Smadar Kedar. A unified framework for explanation-based generalization of partially ordered and partially instantiated plans. *Artificial Intelligence*, 67(1):29–70, 1994.

[54] Henry A. Kautz, David A. McAllester, and Bart Selman. Encoding plans in propositional logic. In *5th International Conference on the Principles of Knowledge Representation and Reasoning*, pages 374–384, 1996.

[55] Henry A. Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *16th International Joint Conference on Artificial Intelligence*, volume 16, pages 318–325, 1999.

[56] Thomas Keller and Patrick Eyerich. A polynomial all outcome determinization for probabilistic planning. In *21st International Conference on Automated Planning and Scheduling*, pages 331–334, 2011.

[57] Phil Kim, Brian C. Williams, and Mark Abramson. Executing reactive, model-based programs through graph-based temporal planning. In *17th International Joint Conference on Artificial Intelligence*, volume 17, pages 487–493, 2001.

[58] Peter Kissmann and Stefan Edelkamp. Solving fully-observable non-deterministic planning problems via translation into a general game. In *32nd Annual German Conference on Advances in Artificial Intelligence*, KI'09, pages 1–8, 2009.

[59] Craig A. Knoblock. Planning, executing, sensing, and replanning for information gathering. In *14th International Joint Conference On Artificial Intelligence*, pages 1686–1693, 1995.

[60] Andrey Kolobov, Mausam, and Daniel S. Weld. ReTrASE: Integrating paradigms for approximate probabilistic planning. *21st International Joint Conference on Artificial Intelligence*, pages 1746–1753, 2009.

[61] Andrey Kolobov, Mausam, and Daniel S. Weld. Classical planning in MDP heuristics: with a little help from generalization. In *20th International Conference on Automated Planning and Scheduling*, pages 97–104, 2010.

[62] Andrey Kolobov, Mausam, and Daniel S. Weld. SixthSense: fast and reliable recognition of dead ends in MDPs. In *24th Conference on Artificial Intelligence*, pages 1108–1114, 2010.

[63] Ron Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[64] Ugur Kuter, Dana Nau, Elnatan Reisner, and Robert P Goldman. Using classical planners to solve nondeterministic planning problems. In *18th International Conference on Automated Planning and Scheduling*, pages 190–197, 2008.

[65] Jonas Kvarnström, Patrick Doherty, and Fredrik Heintz. A temporal logic-based planning and execution monitoring system. *18th International Conference on Automated Planning and Scheduling*, 19(3):198–205, 2008.

[66] Jerome Lang, Paolo Liberatore, and Pierre Marquis. Conditional independence in propositional logic. *Artificial Intelligence*, 141(1-2):79–121, 2002.

[67] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.

[68] Solange Lemai and Felix Ingrand. Interleaving temporal planning and execution: IxTeT-eXeC. In *ICAPS Workshop on Plan Execution*, 2003.

[69] Hector J Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B Scherl. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1-3):59–83, 1997.

[70] Steven J. Levine. Monitoring the execution of temporal plans for robotic systems. *Master's Thesis*, 2012.

[71] Iain Little and Sylvie Thiébaux. Probabilistic planning vs replanning. In *ICAPS Workshop International Planning Competition: Past, Present and Future*, 2007.

[72] Robert Mattmüller, Manuela Ortlieb, Malte Helmert, and Pascal Bercher. Pattern database heuristics for fully observable nondeterministic planning. In *20th International Conference on Automated Planning and Scheduling*, pages 105–112, 2010.

[73] Christian Muise, J. Christopher Beck, and Sheila A. Mcilraith. Flexible execution of partial order plans with temporal constraints. In *23rd International Joint Conference On Artificial Intelligence*, pages 2328–2335, 2013.

[74] Christian Muise, Sheila A. McIlraith, and J. Christopher Beck. Monitoring the execution of partial-order plans via regression. In *22nd International Joint Conference On Artificial Intelligence*, International Joint Conference On Artificial Intelligence, pages 1975–1982, 2011.

[75] Christian Muise, Sheila A. Mcilraith, and J. Christopher Beck. Optimization of partial-order plans via MaxSAT. In *ICAPS Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, COPLAS, 2011.

[76] Christian Muise, Sheila A. McIlraith, and J. Christopher Beck. Improved nondeterministic planning by exploiting state relevance. In *22nd International Conference on Automated Planning and Scheduling*, The 22nd International Conference on Automated Planning and Scheduling, pages 172–180, 2012.

[77] Christian Muise, Sheila A. McIlraith, and J. Christopher Beck. Optimally relaxing partial-order plans with MaxSAT. In *22nd International Conference on Automated*

Planning and Scheduling, The 22nd International Conference on Automated Planning and Scheduling, pages 358–362, 2012.

[78] Nicola Muscettola, Paul H. Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *6th International Conference on Principles of Knowledge Representation and Reasoning*, pages 444–452, 1998.

[79] Bernhard Nebel, Yannis Dimopoulos, and Jana Koehler. Ignoring irrelevant facts and operators in plan generation. In *4th European Conference on Planning*, pages 338–350, 1997.

[80] Hector Palacios and Hector Geffner. From Conformant into Classical Planning: Efficient Translations that May Be Complete Too. In *17th International Conference on Automated Planning and Scheduling*, pages 264–271, 2007.

[81] Fabio Patrizi, Nir Lipovetzky, and Hector Geffner. Fair LTL Synthesis for Non-Deterministic Systems using Strong Cyclic Planners. In *23rd International Joint Conference On Artificial Intelligence*, 2013.

[82] Ola Pettersson. Execution monitoring in robotics: a survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.

[83] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[84] Martin L Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

[85] Miquel Ramirez, Nitin Yadav, and Sebastian Sardina. Behavior Composition as Fully Observable Non-Deterministic Planning. *23rd International Conference on Automated Planning and Scheduling*, pages 180–188, 2013.

[86] Raymond Reiter. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. The MIT Press, 2001.

[87] Jussi Rintanen. Regression for classical and nondeterministic planning. In *18th European Conference on Artificial Intelligence*, pages 568–572. IOS Press, 2008.

[88] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. Partial weighted MaxSAT for optimal planning. In *11th Pacific Rim International Conference on Artificial Intelligence*, pages 231–243, 2010.

[89] Stuart J. Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Prentice hall, 2009.

[90] Scott Sanner and Craig Boutilier. Practical linear value approximation techniques for first-order MDPs. In *22nd Conference on Uncertainty in Artificial Intelligence*, 2006.

[91] Julie Shah, John Stedl, Brian C. Williams, and Paul Robertson. A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In *17th International Conference on Automated Planning and Scheduling*, pages 296–303, 2007.

[92] Fazlul Hasan Siddiqui and Patrik Haslum. Block-structured plan deordering. In *Australasian Conference on Artificial Intelligence*, pages 803–814, 2012.

[93] David E Smith, Jeremy Frank, and Ari K Jónsson. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15(1):47–83, 2000.

[94] Tran Cao Son and Phan Huy Tu. On the Completeness of Approximation Based Reasoning and Planning in Action Theories with Incomplete Information. In *10th International Conference on Principles of Knowledge Representation and Reasoning*, pages 481–491, 2006.

[95] Siddharth Srivastava. *Foundations and applications of generalized planning*. PhD thesis, University of Massachusetts Amherst, 2010.

[96] Manuela M. Veloso, Martha E. Pollack, and Michael T. Cox. Rationale-based monitoring for planning in dynamic environments. In *4th International Conference on Artificial Intelligence Planning Systems*, pages 171–180, 1998.

[97] Thierry Vidal. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental & Theoretical Artificial Intelligence*, 11(1):23–45, 1999.

[98] Richard Waldinger. Achieving several goals simultaneously. *Machine Intelligence*, 8:94–136, 1977.

[99] Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[100] David E. Wilkins. Recovering from execution errors in SIPE. *Computational Intelligence*, 1(1):33–45, 1985.

[101] SungWook Yoon, Alan Fern, and Robert Givan. FF-Replan: a baseline for probabilistic planning. In *17th International Conference on Automated Planning and Scheduling*, pages 352–359, 2007.

[102] SungWook Yoon, Alan Fern, Robert Givan, and Subbarao Kambhampati. Probabilistic planning via determinization in hindsight. In *23rd Conference on Artificial Intelligence*, pages 1010–1016, 2008.

[103] Sungwook Yoon, Wheeler Ruml, J. Benton, and Minh B. Do. Improving determinization in hindsight for online probabilistic planning. In *20th International Conference on Automated Planning and Scheduling*, pages 209–216, 2010.

[104] Hakan L. S. Younes and Reid G. Simmons. VHPOP: versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, 20:405–430, 2003.

[105] Hakan L. S. Younes and Reid G. Simmons. Policy generation for continuous-time stochastic domains with concurrency. In *14th International Conference on Automated Planning and Scheduling*, volume 325, pages 325–334, 2004.

[106] Peng Yu and Brian C. Williams. Continuously relaxing over-constrained conditional temporal problems through generalized conflict learning and resolution. In *23rd International Joint Conference On Artificial Intelligence*, pages 2429–2436, 2013.

[107] Hankz Hankui Zhuo and Subbarao Kambhampati. Action-Model Acquisition from Noisy Plan Traces. In *23rd International Joint Conference On Artificial Intelligence*, pages 2444–2450, 2013.