Optimal Planning with State Constraints

Franc Ivankovic
June 2017

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

Declaration

Except where otherwise indicated, this thesis is my own original work.

Franc Ivankovic
June 2017

Acknowledgements

I am deeply grateful to my supervisor Patrik Haslum, for his support and his limitless patience. I am very thankful for all the discussions that we had and the time spent on reviewing my work.

I also want to thank: Sylvie Thiébaux for all her help and useful suggestions; the other people I worked with on papers, Dana Nau, Miquel Ramirez, and Vikas Shivashankar; other researchers at Data61 and ANU for sharing their knowledge and ideas, including Menkes van den Briel, Enrico Scala, Alban Grastien, Hassan Hijazi, Tommaso Urli, Charles Gretton, Dan Gordon, Phil Kilby, Michael Norrish; my fellow students Josh Bax, Ksenia Bestuzheva, Cody Christopher, Jing Cui, Karsten Lehmann, Boonping Lim, Terrence Mak, Mohammad Abdulaziz, Paul Scott, Fazlul Hasan Siddiqui and Frank Su; and others who worked at ANU or Data61.

Finally, I thank my friends, my family (parents, brothers, grandparents...) and Jessica for their support and encouragement.

Abstract

In the classical planning model, state variables are assigned values in the initial state and remain unchanged unless explicitly affected by action effects. However, some properties of states are more naturally modelled not as direct effects of actions but instead as derived, in each state, from the primary variables via a set of rules. We refer to those rules as state constraints. The two types of state constraints that will be discussed here are numeric state constraints and logical rules that we will refer to as axioms.

When using state constraints we make a distinction between primary variables, whose values are directly affected by action effects, and secondary variables, whose values are determined by state constraints. While primary variables have finite and discrete domains, as in classical planning, there is no such requirement for secondary variables. For example, using numeric state constraints allows us to have secondary variables whose values are real numbers. We show that state constraints are a construct that lets us combine classical planning methods with specialised solvers developed for other types of problems. For example, introducing numeric state constraints enables us to apply planning techniques in domains involving interconnected physical systems, such as power networks.

To solve these types of problems optimally, we adapt commonly used methods from optimal classical planning, namely state-space search guided by admissible heuristics. In heuristics based on monotonic relaxation, the idea is that in a relaxed state each variable assumes a set of values instead of just a single value. With state constraints, the challenge becomes to evaluate the conditions, such as goals and action preconditions, that involve secondary variables. We employ consistency checking tools to evaluate whether these conditions are satisfied in the relaxed state. In our work with numerical constraints we use linear programming, while with axioms we use answer set programming and three value semantics. This allows us to build a relaxed planning graph and compute constraint-aware version of heuristics based on monotonic relaxation.

We also adapt pattern database heuristics. We notice that an abstract

state can be thought of as a state in the monotonic relaxation in which the variables in the pattern hold only one value, while the variables not in the pattern simultaneously hold all the values in their domains. This means that we can apply the same technique for evaluating conditions on secondary variables as we did for the monotonic relaxation and build pattern databases similarly as it is done in classical planning.

To make better use of our heuristics, we modify the $A^\star$ algorithm by combining two techniques that were previously used independently – partial expansion and preferred operators. Our modified algorithm, which we call PrefPEA*, is most beneficial in cases where heuristic is expensive to compute, but accurate, and states have many successors.

Contents

List of Figures

Chapter 1

Introduction

Planning is one of the oldest subareas of artificial intelligence, originating in the early 1960s, with the goal of achieving human-like problem solving capabilities [80, 110]. A planning task consists of an initial state the world is in, a specification of what we want to achieve, or goal, and a list of actions that are available. A solution to the planning problem, called a *plan*, is a sequence of actions that transforms the world from the initial state into one of the goal states.

In the classical planning model, action effects are explicitly given for each of the actions, so determining the way the world is affected by the action is straightforward. When an action is applied, the variables that do not appear in action effects retain the same value as in the previous state. However, some properties of states are more naturally modelled not as direct effects of actions but instead as derived, in each state, from the primary variables according to a set of rules that apply to all states. We call those rules state constraints.

This work focuses on optimal planning with state constraints, which we will employ for a number of purposes. We will use state constraints to elegantly model laws of physics in interconnected systems, such as power networks. In these domains, the planning agent manipulates the system through discrete control actions, such as opening or closing of the switches, yet each action leads to a complicated interaction between elements of the system that depends on the state of the entire network. We will show how usage of state constraints makes some domains, such as controller verification or Sokoban, easier to solve. In order to reason about state constraints, we will integrate planners with systems used to reason about constraints, such as a linear programming solver. We will describe the way we adapted techniques commonly used in planning, namely admissible heuristics, to this setting.

1.1 State constraints

In this section, we will give an overview of general properties of state constraints, describe some ways they can be used and introduce some terms that will appear throughout this thesis.

When using state constraints, we make a distinction between state variables that are directly affected by action effects, called *primary variables*, and the variables whose values are determined by state constraints, called *secondary variables.* Conditions such as goals and action preconditions can depend on both primary and secondary variables. Constraints and secondary variables may take many different forms – for example, secondary variables may be numeric, in which case the constraints may take form of linear inequalities, as we will see in Chapter 3.

Following Helmert's [77] terminology, we refer to the assignment of primary variables as *state* and an assignment of both primary and secondary variables as *extended state.* The assignment of *only* primary variables is simply called a state, or a *reduced state.* The relationship between a state and an extended state depends on the types of constraints that are being used. In some cases, there is always a unique assignment of secondary variables given an assignment of primary variables, in which case there is one extended state corresponding to every state. In other cases, we might have a multiple or an infinite number of extended states corresponding to a single state.

In addition to being used to compute values of secondary variables, introducing state constraints allows us to make a distinction between *valid* and *invalid* states. A valid state is a state in which there is at least one assignment of secondary variables that satisfies all of the constraints exists. In an invalid state, it is not possible to come up with such an assignment. A plan is a sequence of actions that visits only valid states. In this setting, applying an action might not be allowed, not because any of the preconditions are unsatisfied, but because effects of the actions lead to an assignment of values to primary variables that results in an invalid state. Thus, determining whether an action is applicable in a given state does not only involve checking whether the preconditions are satisfied, but also whether the resulting assignment of variables constitutes a valid state.

It should be noted that given our definition of state constraints, we do not necessarily need to have secondary variables in our problem. State constraints might, for example, prohibit some combination of values of primary variables, as is the case in the work of Weld and Etzioni [140]. The notion of state validity, defined through state constraints, is present, yet secondary variables are not required.

1.1.1 Contribution

The two types of state constraints that we will discuss in our work are numeric state constraints and axioms.

- Numeric state constraints allow us to use real numbers as secondary variables. As in classical planning, the primary variables are discrete. For a given assignment of primary variables, there could either be a unique assignment of secondary variables, or there could be multiple or infinitely many such assignments, or there could be none. If there is no assignment satisfying the constraints, the state is invalid.

- Axioms take the form of rules of a logic problem. Unlike in planning with numeric constraints, both primary and secondary variables have discrete domains. Another difference from planning with numeric constraints is that given an assignment of primary variables, a unique assignment of secondary variables satisfying the constraints is guaranteed to exist – i.e. there are no invalid states.

We use state constraints as a construct that enables us to combine classical planning techniques with specialised solvers developed for other types of problems. In the case of numeric constraints, this is a linear programming solver. This widens the types of domains that can be addressed. For instance, numeric constraints allow us to apply planning techniques to problems involving interconnected physical systems. An example that we will return to several times throughout this thesis is power network reconfiguration problem (see Section 3.3.2).

We also show that state constraints can be used to make some domains, previously studied in classical planning, easier to model and solve. With axioms, modelling domains becomes more compact and easier to understand. In some domains, namely Sokoban (Section 5.3.2) and controller verification (Section 5.4.1), use of axioms eliminates some unnecessary choices from the planner therefore reducing the search space.

1.2 Optimal planning

We are interested in performing optimal planning on this sort of problems. Optimal planning is concerned with not just finding any plan that ends in some goal state, but to find the cheapest plan, evaluated according to some cost function.

The reason for this is that in many of the domains that we consider, it is beneficial to have plans that are as cheap as possible. In power network reconfiguration, this is because leaving portions of the network without power is economically costly for power companies, so the process needs to be completed as quickly as possible. In the controller verification domain, where finding a plan is equivalent to finding a fault in the controller, generating optimal plans is not necessary. However, plans that are cheaper, and therefore shorter and simpler, are easier to understand. This makes fixing the faults in the controller simpler as well.

The technique commonly used in optimal planning is heuristic state-space search. This means that a search algorithm, such as A* [111] is used together with a heuristic that determines which nodes (which represent states of the planning problem) to expand. The heuristic estimates cost of reaching the nearest goal state from the state being evaluated. A heuristic that never overestimates this cost is known as *admissible* heuristic. The interest in admissible heuristics is due the fact that there are search algorithms, such as already mentioned A*, which are guaranteed to return an optimal plan if the heuristic has this property.

Domshlak and Helmert [79] give an overview of heuristics used in classical planning. They categorise the heuristics as based on one of four ideas: delete relaxation, critical paths, abstractions and landmarks. In this work, we use and adapt h_{max} and h^+, which belongs to the first category, the pattern database heuristic, which is a form of abstraction, and our disjoint landmark heuristic (which combines ideas from delete relaxation and landmarks). As we are interested in optimal planning we limit ourselves to admissible heuristics.

Most of the time we will deal with cost functions that only depend on actions – that is, actions have constant costs regardless of the state in which they are applied. The cost of the plan is then simply the sum of the costs of the actions making up the plan. In most domains that we will consider, all the actions are assigned equal cost and the objective simply becomes minimising the plan length.

In some cases, however, the cost of an action is a function of the state in which the action is applied. In power network reconfiguration, for instance, leaving greater portions of the network without power is more costly. Until recently (see, for example, work by Geißer et al. [58,60]), planning with such state dependent action costs received little attention. One reason for this is that, while computing state dependent cost is is trivial in forward search, accounting for them in heuristics is more challenging [60].

1.2.1 Contribution

While axioms and various forms of numeric state constraints have been investigated by other researchers [1], optimal planning in this setting has been neglected.

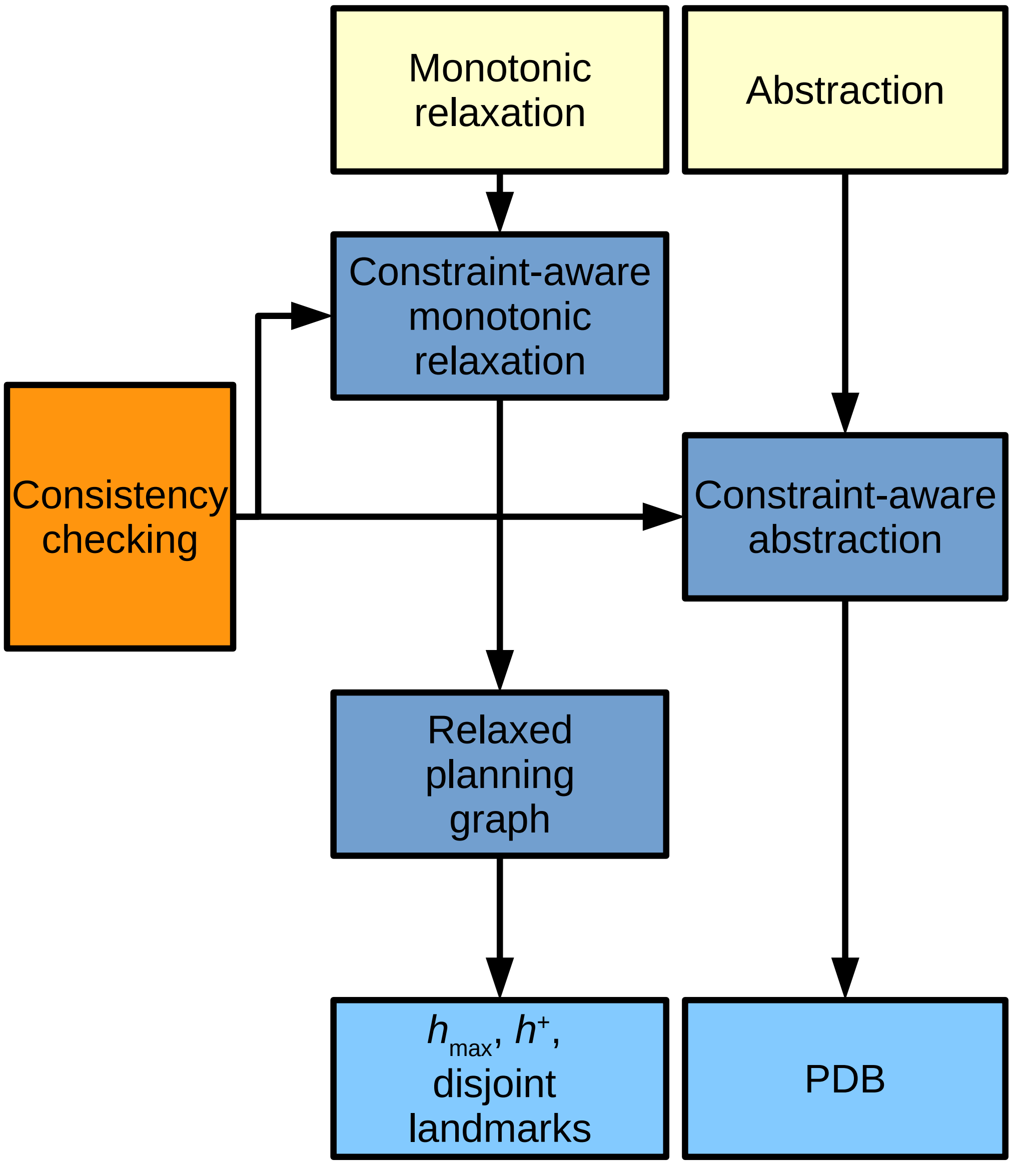
Our contribution is adapting the admissible heuristics used in classical planning to this setting. We base our approach on the idea of monotonic relaxation (Section 2.2.1). The idea is that in a relaxed state, each variable assumes a set of values instead of just a single value. The central challenge is to evaluate the conditions (i.e. action precondition or goals) that involve secondary variables in a relaxed state. This is approached as a consistency checking problem – using consistency checking techniques we determine whether there exists an assignment of secondary variables that satisfies the constraints and the condition that we are testing. In our work with numerical constraints, we use linear programming. With axioms, we use answer set programming and three value semantics.

The diagram in Figure 1.1 summarises our approach. The ability to evaluate conditions on secondary variables allows us to formulate constraint-aware monotonic relaxation, which allows us to build a relaxed planning graph, hence allowing us to compute the h_{max} heuristic. The relaxed planning graph also provides us with relaxed reachability testing, which we use together with the iterative landmark algorithm [74] to compute h^+. Modifying this algorithm to generate disjoint landmarks also enables us to compute a weaker heuristic, equivalent to LM-cut for unit cost actions.

Projection (Section 2.2.3) is an abstraction that works by removing some subset of (primary) variables from the problem. The remaining (primary) variables are called a pattern. This type of abstract state can be though of as a relaxed state in which the variables in the pattern have a single value, while the primary variables not in the pattern simultaneously hold all the values in their domain. We can then employ the same consistency checking procedures as before to evaluate the conditions involving secondary variables. We use the abstraction of the problem to compute pattern databases heuristics (PDBs). We do this in a very similar way as it done in planning without state constraints.

Besides adapting heuristics for problems with state constraints, we also investigated planning with state dependent action costs. We discuss the issues arising from having a cost function that is dependent on an extended state. We adapt the h^+ heuristic to deal with state-dependent action costs.

[1]For more information on axioms see Théibaux et al. [136] and Chapter 2. While our formulation of numeric constraints differs from that of other authors, an overview of related work will also be given Chapter 2.



Monotonic relaxation
Abstraction
Constraint-aware monotonic relaxation
Consistency checking
Constraint-aware abstraction
Relaxed planning graph
h_{max}, h^+, disjoint landmarks
PDB

Figure 1.1: Deriving heuristics for planning with state constraints.

We also modified the A* algorithm by combining two techniques that were previously used independently – partial expansion and preferred operators. The requirement is that the heuristic that we are using needs to return a list of preferred operators – in case of h^+, this is a set of actions that make up the optimal relaxed plan. Our technique, which we call PrefPEA*, is most beneficial in cases where heuristic is expensive to compute, but accurate, and states have many successors. This applies to, for example, our power network reconfiguration domain, there are as many successors to each state as there are switches on the network.

1.3 Thesis Outline

This thesis is structured as follows:

- Chapter 2 will give an overview of background and related work. It will introduce the definitions and the notation that we will use throughout this thesis. We will then explain the techniques used in classical planning that we adapted for planning with state constraints. We will also give an overview of the related work in planning with state constraints.

- Chapter 3 deals with a specific type of state constraints, namely numeric state constraints. We will show how we adapted heuristics introduced in Chapter 2 to this setting and present the domains that we used in our experiments.

- Chapter 4 will investigate state-dependent action costs in optimal planning with numeric constraints.

- Chapter 5 will cover optimal planning with axioms – again, we will show how we adapted the well-known heuristics to this setting and present domains that can be modelled using axioms. We will demonstrate that axioms can make certain domains both easier to model and easier to solve.

- Chapter 6 deals with an algorithm that we developed to improve A* search when an informative, but expensive heuristic is used. We will discuss the related work, present the PrefPEA* algorithm and the experimental results.

- Chapter 7 serves as a conclusion.

1.4 List of publications

Parts of the material in this thesis has previously appeared in the following papers:

- Franc Ivankovic, Patrik Haslum, Sylvie Thiébaux, Vikas Shivashankar, and Dana S. Nau. Optimal planning with global numerical state constraints. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*, 2014.

- Franc Ivankovic and Patrik Haslum. Optimal planning with axioms. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1580–1586, 2015.

- Patrik Haslum, Franc Ivankovic, Miquel Ramirez, Dan Gordon, Sylvie Thiébaux, Vikas Shishankar, and Dana S. Nau. Extending classical planning with state constraints: Heuristics and search for optimal planning. *J. Artif. Intell. Res. (JAIR)*, 2018 (to appear).

Chapter 2

Background and related work

In the previous chapter we mentioned that introduction of state constraints leads to a distinction between primary and secondary variables. While a (very few) planners capable of dealing both with hybrid (a mix of real-valued and discrete) state variables and global constraints exist, in this thesis we will focus on cases where the primary variables have discrete and finite domains. In that sense, our work either meets the definition of classical planning or is only a small step away from it. To generate optimal plans we adapt techniques that were proven successful in optimal classical planning, namely state-space search guided by an admissible heuristic. In this chapter, we will give a definition of classical planning and describe the admissible heuristics commonly used in this setting. This chapter will also give an overview of the related work.

The organisation of the chapter is as follows. Section 2.1 gives a definition of classical planning and highlights some of its limiting assumptions. Terminology and notation used in classical planning will be defined in this section. In subsequent chapters we will build on the formalism introduced here to formally describe planning with state constraints. In section 2.2 we will focus on techniques used for optimal planning in the classical setting. We will discuss heuristics based on monotonic relaxation and abstractions. Section 2.3 focuses on existing work on planning with state constraints. This will cover their use in domains with discrete-valued variables, their use in modelling interconnected physical systems and hybrid domains. Section 2.4 discusses the relation of state constraints to semantic attachments and planning modulo theories (PMT).

2.1 Classical planning

Planning requires a formal statement, or model, of the problem. Here we will discuss the classical planning model first and then highlight some of its limiting assumptions. We will discuss how these assumptions relate to our work.

Planning is a state-transformation problem. A planning problem consists of a finite number of *variables*, each associated with a *domain* of possible values. A *state* is a full valuation over the variables. *Actions* assign new values to a subset of variables and therefore cause transitions between the states. Additionally, planning problems have a *goal* which is usually defined as a valuation over a subset of variables. A *plan* is a sequence of actions that transforms an initial state into a state in which the goal is satisfied. A *planning agent* is required to come up with such a sequence of actions.

The type of planning that has been most extensively studied is *classical planning*. This area deals with a deterministic, static, finite and fully observable state-transition system with restricted goals and implicit time. While classical planning has been a very active and prolific field of research, its limiting assumptions preclude us from dealing with real-world domains that cannot be modelled this way [83]. As these restrictions often make formulation of real world problems either impractical or impossible, there has been an interest in relaxing them or removing them. The following list of assumptions in classical planning is adapted from Ghallab et al. [107]. In classical planning:

1. The set of states is *finite*. For the state-space to be finite, every variable that we are dealing with needs to have a finite domain. Removing this restriction is necessary when dealing with numerical state variables. Real-world domains might require us to consider continuous variables such as time, velocities, positions, and keeping track of resources such as money or fuel [32]. A number of planners that we will discuss in Sections 2.3 and 2.4 are capable of dealing with numeric state variables. For a detailed overview see Coles et al. [32].

2. States are *fully observable*, meaning that our planning agent knows the accurate value for each of the variables. This assumption has been removed to study cases where states are *partially observable*. Partial observability can be modelled by dealing with sets of states rather than single states. Planning under partial observability has been investigated by, for example, Bonet and Geffner [15], Hoffmann and Brafman [84] and Kaelbling et al [89].

3. The system is *deterministic*. Applying an action in a given state brings the system to a single other (predetermined) state. In contrast, by *planning under uncertainty*, we mean domains in which applying an action may lead to a number of different states (and which state we end up in is non-deterministic). A way to deal with this sort of problems is *contingency planning*, meaning that some branches are executed conditionally, based on the outcome of the sensory actions. Techniques that have been employed to deal with non-determinism include Markov Decision Processes (MDPs) [22,37,88] and planning as model checking [8]. If Assumption 2 is removed as well, this leads to further difficulty as the system does not know exactly the current state of the system at run time. Dealing with this sort of problems is called *conformant planning* [67, 131]. The aim becomes to develop non-conditional plans that do not rely on sensory information, but still succeed no matter which state the world is actually in. For an overview of approaches to conformant planning see Palacios and Geffner [112].

4. The system is *static*, meaning that it stays in the same state until an action is applied. If a non static (or dynamic) system is deterministic and fully observable, it can easily be mapped into the static system – i.e. the planning agent knows which *events* will occur in any given state and how the occuring events alter the values of variables. It then simply becomes a modelling choice whether these changes will be described as effects of actions or effects of deterministic events, so relaxing this assumption is not interesting on its own. PDDL+ [50] is an exaple of a modelling language that allows for events and processes controlled by nature, while TM-LPSAT [129] and COLIN [32] are examples of planners that can deal with those features.

5. The planner handles only *restricted goals*, meaning that the objective is to find any sequence of actions that ends in a goal state. *Extended goals* means that we put restrictions not only on the final state, but also on states visited by a plan – this means, for example, specifying states that must be visited, states to be avoided, values that must be maintained once achieved etc. Work on preventing plans from visiting some states by Weld and Etzioni [140] that we will discuss in Section 2.3.3 is one of the simple examples. We will discuss the meaning of state validity in our setting in Chapter 3. It should be noted that in many cases, domains with extended goals can be reformulated as classical planning domains. [1]

[1]The authors mention functions to be optimised as an example of an extended goal,

6. A solution to a planning problem is a linearly ordered finite sequence of actions. Relaxing this assumption is often necessary when some of the other assumptions are relaxed (e.g. when we are dealing with non-deterministic systems). Removing this restriction enables us to present solutions with “richer” mathematical structure. One example is already mentioned contingency planning. Alternatively, a solution might be a partially ordered set of actions or a sequence of sets. This sort of solutions are generated by partial order planners such as UCPOP [113].

7. Time not explicitly defined – actions are instantaneous state transitions. Plan consists of a sequence of actions, but we are not concerned how long does each of the actions take (or, if we are, this can be encoded as action cost). In *temporal planning*, action duration and concurrency are taken into account. For history and an overview of planner capable of dealing with temporal planning problems see Coles et al. [31]. The interest this area increased when PDDL [49] was extended (as PDDL2.1) to include temporal features. Some of the early (pre-PDDL2.1) planners include IxTeT [65], TLplan [3], TALplanner [38] and Zeno [114]. More recent work includes LGP-td [63], Crickey [33] and Colin [30].

8. Planning takes place *offline*. No changes occur in the system while the agent is coming up with the plan. In practical applications the planner often has to deal with an evolving system, which may also be partially observable or non-deterministic. In such cases, the planner must check online whether the solution it came up with remains valid, and if it doesn't, revise it (or re-plan). As online planning is related to partial observability and non-determinism, contingency planning and conformant planning approaches are often utilised. Ross et al. [124] describe use of POMDP to solve this kind of problems.

In this work, we will keep all of these assumptions apart from Assumption 5. The reason for this is that in planning with numeric constraints we make a distinction between valid and invalid states. Additionally, in Chapter 4, we discuss cost functions which depend on the extended state. While we are adding mechanisms to deal with numeric variables, state variables remain finite, so our state spaces remain finite as well, respecting 1. We keep the Assumption 7 – for example, in our power network domain (Section 3.3.2), we assume that the time between each switching action is long enough for

which means that under their definition optimal planning violates this assumption. However, optimal planning is usually considered classical if it respects all of the other assumptions.

the system to reach a stable state. Similarly, in long-haul transportation domain (Section 3.3.3), we simply assume that trucks do one trip a day. The advantage of keeping most of the assumptions of classical planning is that we can easily adapt many of the techniques developed for classical planning, while still solving problems which would be computationally difficult using classical planning.

Planning with axioms (Chapter 5) respects all of the above listed assumptions. However, we will occasionally use the term classical planning to mean planning without state constraints, even though this use might not be entirely correct.

2.1.1 Finite domain representation

While there are a number of formalisms defining the classical planning problem [107], here we will use the notation and the definitions from Domshlak and Nazarenko [39] called finite-domain representation (FDR), which is based on the SAS+ formalism [21]. Unlike some earlier formalisms for classical planning, such as STRIPS [48], FDR uses multi-valued state variables instead of propositional atoms (this feature was inherited from SAS+).

State

As explained above, one of the assumptions of classical planning is that the set of states is finite. In FDR, this is equivalent to stating that in a planning problem, we have a finite set of state variables V, with each $v \in V$ being associated with a finite domain $\mathcal{D}(v)$. Here we define a *partial variable assignment* and a *state.*

Definition 1. *V is a set of state variables, with each $v \in V$ being associated with a finite domain $\mathcal{D}(v)$. A partial variable assignment p is a function of a variable subset $\mathcal{V}(p) \subseteq V$ that assigns each $v \in \mathcal{V}(p)$ a value $p[v] \in \mathcal{D}(v)$ from its domain. A partial variable assignment s is called a* state *if $\mathcal{V}(s) = V$ (that is, a state assigns every variable a value in its domain).*

We refer to a partial variable assignment over a single variable as an *elementary formula.*

Partial variable assignments are used to encode conditions on states, which are used as *goals* and *action preconditions.* Conditions are defined as follows:

Definition 2. *Given a partial variable assignment p and a state s, the value of the condition $s[p]$ is*

- *true if* $s[v] = p[v]$ *for all* $v \in \mathcal{V}(p)$ *and*
- *false otherwise.*

If $s[p]$ *is true, we say that* p holds *in* s*.*

An empty partial variable assignment $p_\emptyset$ holds in every state.

Planning problem

The definition of the planning problem is given by:

Definition 3. *[Adapted from Domshlak and Nazarenko [39]] A planning task in FDR representation is a tuple* $\Pi = \langle V, A, s_0, G, cost\rangle$ *where*

- V *is a set of finite-domain state variables.*
- s_0 *is the initial state.*
- G *is the* goal*, which is a partial variable assignment over* V*.*
- A *is a finite set of actions. Each action* $a \in A$ *is a pair* $\langle \mathrm{pre}(a), \mathrm{eff}(a)\rangle$*, where*
  - $\mathrm{pre}(a)$ *is action's* preconditions
  - $\mathrm{eff}(a)$ *is action's* effects

  Both action preconditions and action effects are partial variable assignments.
- $cost(a, s)$ *is a cost function. The function takes an action and a state as an input and returns a non-negative real number which represents the cost of applying the action in the given state.*

Actions and plans

An action a is *applicable* in a state s iff its precondition holds in state s. Application of a in s changes the values of every $v \in \mathcal{V}(\mathrm{eff}(a))$ to $\mathrm{eff}(a)[v]$ and we denote the resulting state by $s[\![a]\!]$. All of the other variables retain the same value as in s. Formally, this is expressed as

$$s[\![a]\!][v] = \begin{cases} \mathrm{eff}(a)[v] & \text{if} \quad v \in \mathcal{V}(\mathrm{eff}(a)) \\ s[v] & \text{otherwise.} \end{cases}$$

By $s[\![\langle a_1, ..., a_k\rangle]\!]$, we denote a state that is obtained by sequentially applying the actions $a_1, ..., a_k$ (provided that all the actions are applicable in the state obtained by applying the preceding action), starting from state s.

Definition 4. *Given a problem* $\Pi = \langle V, A, s_0, G, cost \rangle$ *and a state* s*, a sequence of actions* $a_1, ..., a_k$ *is called an* s-plan *if the goal holds in* $s[\![\langle a_1, ..., a_k \rangle]\!]$*. The* cost *of an s-plan is the sum of the costs of the actions that the plan consists of.*

$$cost([\![\langle a_1, ..., a_k \rangle]\!], s) = \sum_{i=1}^{k} cost(a_i, s[\![\langle a_1, ..., a_i \rangle]\!])$$

An s-plan is considered *optimal* if its cost is minimal among all s-plans. Finding an optimal plan from an initial state, or s_0-plan, is called *optimal planning.* A special case which we will consider is when all actions are assigned equal and constant cost (i.e. cost is always the same regardless which action is applied in which state) and the objective becomes minimising the plan length.

Most planning research has focused on cases where cost is only a function of the action, rather than an action and a state. We will, however, also consider domains with *state-dependent action costs*, where the cost of an action varies depending on the state in which the action is applied. State dependent action costs will be the subject of Chapter 4.

2.2 Techniques for optimal planning

A common technique used in optimal planning is state-space heuristic search. *Heuristic functions*, in general, estimate the cost of reaching the "end state" (in planning, some goal state) from a given state and are used to guide informed search algorithms [80]. *Admissible* heuristics are lower bound functions – a heuristic is admissible iff it never overestimates the true cost. Admissible heuristics are used in optimal planning because certain optimal search algorithms, like A* [111], guarantee that the solution returned is optimal, provided that the heuristic is admissible. In such cases, efficiency of the heuristic search depends on the accuracy of the heuristic function. The closer the estimate is to the true optimal cost, the less search is required to find and prove the optimality of a solution. Here, we will give an overview of several admissible heuristics. In subsequent chapters we will show how these heuristics were adapted for planning with state constraints.

2.2.1 Monotonic Relaxation

The standard approach to deriving admissible heuristics is to define relaxed version of a problem, as cost-optimal solution to the relaxed problem is a lower bound on the cost of the optimal plan (and is therefore an admissible heuristic cost estimate). One such relaxation is *monotonic relaxation.* Here we will present monotonic finite-domain representation (MFDR). For binary variables MFDR is equivalent to *delete relaxation*[2] and can be considered its generalisation to finite-domain variables[3]. According to Domshlak and Nazarenko [39], it is not clear to whom should the original idea of monotonic relaxation for multi-valued variable domains be attributed, but it can be traced back at least to the work of Helmert [77] on the Fast Downward planning system.

Definition 5. Relaxed planning task *is defined by a tuple* $\Pi^+ = \langle V, A, s^+, G, cost\rangle$.

Apart from using a relaxed state s^+ instead of non-relaxed state s_0, this is the same as FDR. The rules for evaluating whether a partial variable assignment holds and for applying an action are, however, different. The key distinction that makes MFDR a relaxed version of the problem is that a variable can have multiple values at the same time. As actions are applied, variables *accumulate* values rather than switching between them.

Definition 6. *In MFDR a* relaxed state s^+ *assigns each variable* $v \in \mathcal{V}$ *a (non-empty) subset of values from its domain,* $s^+[v] \subseteq \mathcal{D}(v)$.

Given any FDR state s, we can obtain a relaxed state s^+ by simply replacing each assignment of a value to a variable $v_i = x_i$, with a set containing only that value, $v_i = \{x_i\}$, for all variables in V. Computation of all of the heuristics that we will describe in the next section starts by creating a relaxed state from the state for which we want to compute the heuristic value.

The relaxed state represents a set of states, namely those obtainable by assigning each variable v_i one value from its value set $s^+[v_i]$:

$$\text{states}(s^+) = \{\{v_1 = x_1, \dots, v_n = x_n\} \mid \forall i : x_i \in s^+(v_i)\}$$

Given a partial variable assignment p, $s^+[p]$ denotes the set of values that p can take in s^+.

Definition 7. $s^+[p] = \{s[p] \mid s \in \text{states}(s^+)\}$

[2]The idea of using delete relaxation originated for domain independent planning orginiated from Blum and Furst [10]. Bonet, Loerincs and Geffner [17] used the delete relaxation to create an explicit heuristic.

[3]For planning with binary variables, the relaxed planning task is typically defined by changing the action set, rather than redefining the effects of actions on sets of states [57]. The definition that we are using here (borrowed from Domshlak and Nazarenko [39]) is, however, easier to generalise to planning with state constraints.

In other words, $true \in s^+[p]$ if and only if there exists a state $s \in$ states(s^+) such that $s[p] = true$ (and analogously for *false*). A condition p holds in a relaxed state if it is true in at least one of the states in the set.

Recall that goals and action precondition are conditions (that is, a partial variable assignment). Applicability of an action and action effects are modified in a relaxed setting in the following way:

Definition 8. *An action a is* applicable *in a relaxed state s^+ iff its precondition holds in s^+. Application of an action changes values of variables in the action's effects $v \in \mathcal{V}(eff(a))$from $s^+[v]$ to $s^+[v] \cup \{eff(a)[v]\}$.*

As actions are applied, the set of values associated with each variables grows – applying an action can add a new value to the set of values, but it cannot remove a value. Hence the name *monotonic* relaxation.

An MFDR action sequence $\langle a_1, ..., a_K \rangle$ applicable in a relaxed state s^+ is an s^+-plan if $G[v] \in s^+[\![\langle a_1, ..., a_k \rangle]\!]$ for all $v \in \mathcal{V}(G)$. A plan for Π^+ starting from a relaxed state s^+ is called a *relaxed s^+-plan.*

The idea of using delete relaxation for domain independent planning originates with work by Bonet, Loerincs and Geffner [17]. Starting with Graphplan [11], HSP [16] and FF [85], heuristics based on delete relaxation became common in many planning systems. The heuristics employed, HSP and FF were, however, inadmissible variants of relaxed reachability heuristics, so those systems performed non-optimal planning. The next section deals with deriving admissible heuristics from MFDR.

2.2.2 Relaxation-based heuristics

Admissible heuristics built using monotonic relaxation that we will discuss here are h^+, h_{max} and LM-cut. These heuristics were first formulated for the delete relaxation, but work with MFDR formulation as well.

Computing any of those heuristics starts with creating a MFDR. Given a planning task $\Pi = \langle V, A, s_0, G, cost \rangle$ and a state s of Π for which we want to compute the heuristic cost estimate, we create the relaxed planning problem $\Pi^+ = \langle V, A, s^+, G, cost \rangle$, where s^+ is the relaxed state obtained from s.

Optimal delete-relaxed plan and $h^+(s)$ heuristic

Definition 9. *For any state s of* Π, *the* optimal relaxation heuristic $h^+(s)$ *is defined as the cost of an optimal relaxed s^+-plan for MFDR task $\langle V, A, s^+, G \rangle$.*

While this heuristic is the strongest possible heuristic based on delete relaxation [9], (i.e. $h_{max}(s^+) \leq h^+(s^+)$ and $h^{LM\text{-}cut}(s^+) \leq h^+(s^+)$ for all states in every domain), unfortunately, it is NP-equivalent to compute [20]. Other admissible heuristics based on delete relaxation are therefore trying to find a lower bound on h^+ as close as possible to its actual value in a computationally cheaper way. For this reason, it is often desirable to find h^+ in order to assess how close are the other heuristics to “the holy grail they seek” [74].

Besides using h^+ as a heuristic, finding optimal delete-free plans is desirable in domains in which actions don’t have any delete effects. Examples include the minimal seed-set problem from systems biology [55] and relational database query plan generation [123].

A number of methods for computing h^+ have been developed. One possible approach is to remove the delete effects from actions and treat computing h^+ as any other planning problem, as it was done by Helmert and Domshlak [79]. This is, however, not very efficient and in their experiments leads to many instances where the heuristic value could not be computed. (The authors did not propose using this method for guiding the heuristic search or optimal delete-free planning. They wanted to find out how close the other heuristics based on delete relaxation that they considered were to h^+.) Fukunaga and Imai [87] propose an integer programming approach to computing h^+. Pommering and Helmert [117] use branch-and-bound and IDA* and incrementally computed LM-cut heuristic, as well as exploiting some other properties of delete-free planning, to compute optimal delete-free plans. Gefen and Brafman’s [56] method consists of identifying fact landmarks and then pruning the search space using a number of techniques that benefit from the obtained information. The method for computing h^+ that we used in our implementations will be discussed in Section 3.5.4.

Although h^+ is computationally expensive, it provides us with more information about the state being evaluated than just heuristic cost estimate – it gives a set of actions that make up the optimal relaxed plan. In Chapter 6, we will show how this additional information can be used together with some alterations to A* to reduce the number of nodes evaluated during search.

Building a relaxed planning graph and calculating the $h_{\max}$ heuristic

As already stated, given that h^+ is computationally expensive, cheaper heuristics that try to approximate h^+ have been developed. One such (admissible) approximation is h_{max} heuristic, which can be computed in several different ways. Here we will explain how it can be computed by building a *relaxed planning graph.*

Besides enabling us to compute the h_{max} heuristic, building a relaxed graph provides us with a *relaxed reachability test.* Given a state s and a subset of actions of Π, $A' \subseteq A$, building a relaxed planning graph using only actions in A' tells us whether the goal is relaxed reachable from s using only A'. We will use relaxed reachability tests in our implementation of h^+ (Section 3.5.4) and disjoint landmark algorithm (Section 3.5.5).

The explanation that we will present here is adapted from Halsum [71]. The relaxed planning graph consists of alternating layers of actions and relaxed states. Given a state s for which we want to find $h_{max}(s)$, the first relaxed state is obtained by creating the corresponding relaxed state s^+. The first layer of actions consists of all actions that are applicable in the initial relaxed state, $a_1, ..., a_k$. The next layer is a relaxed state in which for every variable the set of values consists of the union of the sets of values that are obtained by applying all the actions in the first layer of actions, $s[v] \cup \{\mathit{eff}(a_1)[v]\} \cup ... \cup \{\mathit{eff}(a_k)[v]\}$. Subsequent layers of actions and relaxed states alternate until we reach a state in which the goal is satisfied or until we run out of applicable actions. If we run out of applicable actions, the goal is unreachable (and if the goal is unreachable in the relaxed setting, it implies that it is unreachable from this state in the non-relaxed case as well).

Since the relaxed state contains initial values of variables, they can be said to be reachable in zero steps. Values in the second relaxed state are reachable within one step. The relaxation lies in the fact that all of the values in the second relaxed state cannot be reached at the same time, since in non-relaxed setting a variable can have only one value. Additionally, values of a variable added in the second relaxed state for two different variables might not be achievable at the same time, as actions assigning those values might be incompatible. In the third relaxed state (as well as all the subsequent relaxed states), it is not even certain whether the values appearing are actually reachable. However, if a value is *not* found in the n-th relaxed state, then it is certain that it cannot be reached in $n - 1$ steps. Thus, the index of the relaxed state in which the value appears is the lower bound on the number actions needed to reach it [71]. If all of the actions have equal and constant cost c, then the cost of reaching a relaxed state s^+ is $\text{cost}(s^+) = c(n - 1)$. (The cost of the first state in a relaxed planning graph is zero.)

Definition 10. *Given a state s of a problem Π and a relaxed planning graph starting from s, the heuristic cost estimate $h_{max}(s)$ is the cost of reaching the cheapest relaxed state in which the goal ofΠ has been reached [71].*

This heuristic is a lower bound on h^+ as the optimal relaxed plan cannot contain fewer actions than there are action layers in the relaxed planning graph and the cost of that plan cannot be less than the number of layers of actions times the action cost. In Section 3.5.4, we will present the version of this heuristic for domains where actions have different (but not state-dependent) costs.

LM-cut heuristic

Unfortunately, $h_{\max}$ is not a very accurate heuristic. In a set of experiments by Helmert and Domshlak [79] designed to measure the relative accuracy of a number of different heuristics with the respect to h^+, $h_{\max}$ performed the worst. The heuristic that proved to be the closest to h^+ was the *landmark cut heuristic* (LM-cut). The authors report the average additive errors for $h_{\max}$ and LM-cut as 27.99 and 0.28, receptively. The relative errors were 68.5% and 2.5% repectively. For more than 70% of the instances, LM-cut computed the exact h^+ value [14, 79]. This section will describe the LM-cut heuristic.

LM-cut heuristic was introduced by Helmert and Domshlakh [79] and its computation involves finding a collection of *disjunctive action landmarks.* A disjunctive action landmark, which we denote $\mathcal{L} = \{L_1, \ldots, L_n\}$, is a set of actions in which there is at least one action that must be contained in every plan. The *cost of a landmark* is defined as equal to the cost of the cheapest action in that landmark [117].

Definition 11. *LM-cut heuristic is the sum of costs of landmarks making up collection $\mathcal{L}$:*

$$h^{LM\text{-}cut}(s) = \sum_{L \in \mathcal{L}} min_{a \in L} cost(a)$$

The definitions and the description of the procedure used to calculate LM-cut presented here are adapted from Bonet and Helmert [14] and Helmert and Domshlak [79]. Before describing the algorithm, we will describe *justification graphs.* Computing the justification graph starts with finding the h_{max} for all of the variables. Then, a new planning task Π' is computed by performing two modifications of the relaxed planning task Π^+. First, the goal and all of the operators preconditions are modified by removing all except one variable-value assignment – we only keep the assignment with the highest value of h_{max} (mapping of each of the action to one of its effects is called *precondition-choice function* by some authors [14]). If there are multiple variables that fit this criterion, ties are broken arbitrarily. Bonet and Helmert state that in their experiments they have observed that accuracy of the heuristic varies significantly depending on how the preconditions are modified. Second, each action a is replaced by a number of copies $a_1, \ldots, a_n$ such that if $\text{eff}(a) = \{v_1 = x_1, \ldots, v_n = x_n\}$, then $\text{eff}(a_1) = \{v_1 = x_1\}, \ldots, \text{eff}(a_n) = \{v_n = x_n\}$. After these transformations, all of the actions have a single precondition and a single effect. These transformations do not alter the h_{max} value of the initial state.

The modified problem is then used to create a justification graph for Π'. Justification graph is a directed weighted graph in which the vertices are elementary formulas, and which has an arc from u to v with weight w iff there is an operator with a precondition u, effect v and cost w (parallel arcs are allowed if there are multiple actions with same precondition and same effect). The authors use the term *justification graph*, because, although it describes a planning task much simpler than the original planning task, it retains enough information to *justify* the h_{max} costs. On the justification graph starting from s, the length of the shortest path from s to the goal is $h_{max}(s)$. Labelling the start state s and the goal t, an $s-t$*-cut* is a a partition of vertices into two sets that separate s from t. *Cut-set* is a set of all edges that cross from the set containing s to the set containing t. As paths that from the start s to the goal t traverse at least one arc in the cut-set, it is straightforward to see that every cut-set of a justification graph is a disjunctive action landmark for the planning task.

The steps for finding the collection $\mathcal{L}$ are listed below. The procedure consists of alternately computing a landmark and then modifying the cost function before computing the next landmark. We denote the cost function used in step i as $cost_i$ and the landmark computed in step i as L_i (the cost function calculated in step i is therefore $cost_{i+1}$). Initially, the cost function is the same as the original cost function, $cost_i = cost$. At step i, the landmark L_i and the cost function $cost_i$ are computed using the following steps:

1. Compute h_{max} values for every variable assignment. If $h_{\max}(t) = 0$, terminate.

2. Compute a modified planning task, Π', by performing the transformations explained above. After this step, each action has only one precondition and only one effect.

3. Construct the justification graph G_i.

4. Construct an $s-t$-cut $C_i = \langle {V^0}_i, {V^\star}_i \cup {V^b}_i \rangle$ where ${V^\star}_i$ contains t all of the nodes from which t can be reached through zero cost edges, ${V^0}_i$ contains s and all nodes reachable from s without passing through some nodes in ${V^\star}_i$ and ${V^b}_i$ contains all other nodes.

5. The landmark L_i is a set of labels of the edges (actions) that form the $s-t$-cut C_i (lead from ${V^0}_i$ to ${V^\star}_i$).

6. Let $m_i = \min_{a \in L_i} c_i(a)$. We modify the costs of all actions as

   (a) $cost_{i+1}(a) = cost_i(a)$ if $a \notin L_i$ and

   (b) $cost_{i+1}(a) = cost_i(a) - m_i$ if $a \in L_i$.

Bonet and Helmert [14] note that in case of *binary-cost* planning tasks, where all action costs are limited to 0 and 1, the computed landmarks are disjoint, $L_i \cap L_j = \emptyset$ for all $i \neq j$.

2.2.3 Abstraction-based heuristics

Another group of admissible heuristics are those based on *abstractions*, which includes *pattern database heuristics* (PDBs).

Abstraction in general means to ignore some information or some constraints to make problem easier. In our context, it refers to a mapping of a planning problem to an abstract planning problem that makes fewer distinction between states. Subsets of states are aggregated into one, while making sure that the existence of path between two states implies existence of path of equal or lower cost between the corresponding abstract states [80]. Obviously, the cost of reaching the goal from a given abstract state is a lower bound on reaching the goal in the original problem, so it can be used as heuristic cost estimate. *Size* of the abstraction of the number of abstract states in the abstraction of the planning problem.

PDBs are the most well known and widely used abstraction heuristics [80]. They were introduced by Culberson and Schaffer [35], who used for optimal solving the 15 puzzle. Korf adapted their concept in solving Rubik's cube problem [98], and Korf and Zhang [99] used for finding the optimal global alignment of DNA or amino-acid sequences. While all of those cases depended on manual construction of PDBs, Edelkamp [41] generalised their approach to domain-independent classical planning.

PDB heuristics are based on the idea of *projection* – a set of states are aggregated together if all variables in a subset $V^A \subseteq V$, called the *pattern*, have the same values in all of those states.

Definition 12. *Under* projection*, state s corresponds to an abstract state s^A if and only if they agree on the values of the variables in a chosen pattern.*

Key challenge for PDB heuristics is choosing which variables to keep and which to discard. The methods for finding a good pattern have been discussed by Haslum et al. [72].

Variables not in the pattern are removed from the initial state, goals, action preconditions and action effects. We can view an abstract state s^A as analogous to a relaxed state in which variables in V^A have only a single value and variables not in V^A have all the values in their domain. The abstract state s^A represents a set of states

$$\begin{aligned}\text{states}(s^A) \quad = \quad & \{\{x_1 = v_1, \ldots, x_n = v_n\} \mid \\ & v_i = s^A(x_i) \,\text{if}\, x_i \in A; \text{else}\, v_i \in \mathcal{D}(x_i)\}.\end{aligned}$$

The point of aggregating states together is to turn the problem into one that is small enough to be solved optimally for every state by blind exhaustive search (the size of the abstraction is bounded by $\prod_{v \in A} \mid \mathcal{D}(v) \mid$). A planner that uses PDB heuristic first precomputes the optimal cost of reaching the goal for all abstract states and stores the values in a look-up table (hence the name pattern database heuristic). The planner then uses these values each time the heuristic cost estimate is needed (i.e. given a state s it finds the value for the corresponding abstract state s^A). In classical planning, PDBs can be effectively constructed by an exhaustive reverse exploration from the abstract goal states. This is, however, not easily adapted to problems with state constraints, as we will explain in Sections 3.5.2 and 5.6.4.

Besides PDBs, another class of abstraction heuristics worth mentioning are merge-and-shrink heuristics, which strictly dominate the PDBs. We haven't used them in our work, so we will omit describing them here, but details can be found in Helmert et al. [80].

2.3 State constraints

In the classical planning model that we discussed in Section 2.1, variables are assigned values in the initial state and remain unchanged unless explicitly modified by action effects. However, it is often more natural to model some properties of a state not as direct effects of actions, but as derived, in each state, via a set of rules which apply for all states. We refer to these rules as state constraints and these are the main subject of this thesis. In this section, we will give an overview of general properties of state constraints and list the related work.

When using state constraints, we make a distinction between state variables that are directly affected by action effects, called *primary variables*, and the variables whose values are determined by the state constraints, called *secondary variables*. The domains of secondary variables are related to the form of constraints used – for example, they might be real numbers, in which case the constraints may take form of linear inequalities, as we will see in Chapter 3.

Following Helmert's terminology [77], we will refer to assignment of all of the primary variables as a state and an assignment of all primary and all secondary variables as an *extended state*. The relationship between reduced states and extended states varies depending on the type of state constraints used. For instance, when using axioms, there is always one unique extended state corresponding to each state. With numeric state constraints, there might be one, more than one (possibly infinitely many) or none (if the state is invalid) possible extended states given an assignment of primary variables.

2.3.1 Axioms in PDDL

Axioms in PDDL [136] are an example of state constraints that take the form of rules of a logic program. In this case, both primary and secondary variables have finite and discrete domains, in line with Assumption 1. Hence, planning with axioms is classical planning.

Primary variables are obtained from *basic* predicates and secondary variables are obtained from *derived* predicates. Axioms have the form of rules with the derived variable in the head and the body being a formula built using both basic and derived predicates. They provide us with a natural way to reason about some properties of networks, such as a way of computing transitive closure. This is useful in dealing with real-world structures such as power grids, networks of pipes, traffic flows etc. For example, one benchmark that was used in the deterministic track of 2004 International Planning Competition (IPC-4) [43] is a power supply restoration (PSR) problem [135]. In this problem, we are given a power network consisting of generators, busbars (which are the points where loads connect to the network), power lines and switches and the task of reconfiguring the network by opening and closing the switches. In PSR axioms are used to compute connectivity between different network elements. A more detailed description and more advanced formulation of the problem will be given in Section 3.3.2.

There are a number of early planners that were capable of reasoning with this types of rules, including work by Manna and Waldinger [104], UCPOP by Barret et al. [6], SHOP by Nau et al. [108] and GPT planner by Bonet and Geffner [13,18]. The first version of PDDL [105] included axioms and derived predicates. Unfortunately, they were removed in PDDL2.1 version [49], which was an extension of the PDDL language to temporal planning, and fell into disuse afterwards. A common criticism of axioms was that they were a non-essential language feature [136] – i.e. all domains that can be expressed using axioms can be rewritten without axioms. Several authors (such as Gazen et al. [53], Garagnani [52] and Davidson and Garagnani [36]) held the view that, as axioms are a non-essential language feature, it might be better to compile them away, rather than to deal with them explicitly. The advantage of this is that it allows for using simpler, efficient, already existing planners that don't implement axioms. Another criticism was that their semantics was ill-specified. Specifically, the organisers of the 2002 International Planning Competition objected that the conditions under which the truth of the derived predicates in PDDL could be uniquely determined were unclear [136].

However, Thiébaux et al. [136] proved that any compilation scheme results in either a worst-case exponential blow-up in the size of domain description or worst-case exponential blow up in the size of the length of the shortest plan. The same paper also provides clear semantics for axioms (by using negation as failure and requiring the set of axioms to be stratified), while remaining consistent with the original definition from the first version of PDDL [105]. Axioms were re-introduced in PDDL2.2, which was used in 2004 International Planning Competition (IPC-4). Modern planners that provide support for axioms include FF_X by Thiébaux et al. [136], LPG (Gerevini et al. [64]), Fast Downward (Helmert [77], Marvin (Coles and Smith [34]) and LAMA (Richter and Westphal [121]). However, there is no research on cost optimal planning with axioms, which is a problem that we will focus on in Chapter 5. In the same chapter we will show that axioms allow us to formulate some domains in a way that makes them easier to solve.

2.3.2 Other uses of state constraints in discrete domains

Work by Francès and Geffner [51] models problems with constraints over variables with discrete domains. While there is no state validity or secondary variables, state constraints are still used to encode action preconditions and goals. A constraint might take the form of a conjunction of relations between the variables (i.e. if we the variables have integer values, they might be equalities or inequalities), with a single variable appearing more than once in the constraint.

As an example, they introduce the counters domain. This domain features $\bar{n}$ counters, $X_1, \ldots, X_{\bar{n}}$, each ranging over integers $0, \ldots, \bar{m}$. Actions $\mathsf{inc}(i)$ and $\mathsf{dec}(i)$ increment and decrement, respectively, counter i by 1. Initial values of the counters can be all zero, all maximum, or random. The goal is $X_i < X_{i+1}$ for $i \in [0, \bar{n}-1]$.

Like in our work, one of the central ideas is accounting for state constraints in heuristics. While constraints of this form can easily be compiled away, the authors demonstrate that accounting for them enables us to formulate stronger relaxations. For example, we could create an atom $\mathsf{val}(i,k)$ for each equality $X_i = k$ and an atom $\mathsf{less}(i,j)$ for each inequality $X_i < X_j$. The goal is then represented by $\mathsf{less}(i,i+1)$ for $i \in [0, \bar{n}-1]$ and the initial state by $\mathsf{val}(i,0)$ for all i. Unfortunately, treating the constraints this way prevents us from accounting for them in the monotonic relaxation, due to the that it forces the relaxation to evaluate each of those atoms in isolation from one another. In the initial relaxed state applying the inc action just once to all of the variables makes both $\mathsf{val}(i, 0)$ and $\mathsf{val}(i, 1)$ true for all i in the subsequent relaxed state. This, in turn, makes all of the atoms in the goal true and gives the computed value of h_{max} as 1, (while the actual number of actions that we need to apply to reach the goal is $n(n - 1)/2$). This underestimate can happen whenever goals or action preconditions contain different non-unary atoms involving state variables.

The authors observe that the cause of this is that the value-accumulating relaxation makes two simplification – (i) *monotonicity*, i.e. we add values to the set as we apply actions, and (ii) *decomposition*, which means that the conjunction of atoms is regarded as true in a propositional layer whenever each one of the atoms in the conjunction is true. Their approach is to develop a planning graph, called *constrained relaxed planning graph* (CRPG) that retains monotonicity, but avoids decomposition. This is done by removing the assumption that a variable can have more than one value at the same time in a relaxed state. This lets them formulate constrained versions of h_{max} and h_{FF} [82] heuristics. Unfortunately, computation of these new heuristics is intractable, so the authors also develop weaker approximations using tractable, but incomplete local consistency algorithms.

In Section 3.3.4 we will model the counters domain using switched constraints, but without conditional effects. We will also use the concept of strengthening the relaxation by removing decomposition in developing heuristics both for numeric state constraints and for axioms.

2.3.3 State validity

It should be noted here that introducing state constraints does not necessarily mean that we have a set of secondary variables. Another property of a state that we might be interested in is determining *state validity*, or to make a distinction between *valid* and *invalid* states. As we will show in Chapter 3, these two uses are not mutually exclusive. The same set of constraints can be used to determine the state validity and compute the values of secondary variables as part of the same process.

As an example, we might want to place a restriction on the possible values of two or more variables, such as “at least one of these has to true” or, given variables that represent integers, “one must be greater than the other at all times”. In a domain where we are scheduling jobs for a hotel staff, we might want to express requirements of the form “at least one person must be at reception at all times”. In other words, state constraints can be used to prohibit the plan from visiting the states that we want to rule out. If there are only primary variables, having an invalid state simply means that the given assignment does not satisfy the constraints. If we have secondary variables as well, an invalid state is the one in which we cannot come up with an assignment of secondary variables that satisfies all of the constraints – i.e. there is no extended state corresponding to the given assignment of primary variables. When we have this requirement, applying an action might not be allowed, not because any of the preconditions are unsatisfied, but because effects of the actions lead to an invalid state. Thus, determining whether an action is applicable in a given state does not only involve checking whether the preconditions are satisfied, but also whether the resulting assignment of primary variables respects all of the constraints.

There are various reasons why we might want to impose such restrictions, depending on what types of domains are we dealing with. For instance, we might want to preclude the planner from generating plans that are impossible to execute (because they violate laws of physics) or we might want to avoid visiting some states because they violate some safety constraints. In the PSR example, power lines and generators have limited capacities and changing the configuration of the network might overload some of them. State constraints are used for expressing the laws of physics governing the flow of power and adding bounds related to capacities of the devices. They can then be used to determine whether the given configuration of switches is allowed, given those limitations.

The need to prohibit planners from generating plans that visit certain states has been discussed by Weld and Etzioni [140], somewhat inspired by Asimov's three laws of robotics. The authors discuss different ways of making sure that the planner does not disturb certain facts. In their case, there are no secondary variables and the constraints are used to place restrictions on the primary variables. The authors define *dont-disturb* primitive that takes a single, function free sentence as an argument. For example, to prevent a planning agent from deleting files that are not backed up on a tape, the following constraint is used: $\mathsf{dont\text{-}disturb}(\mathsf{written\text{-}to\text{-}tape}(f) \vee \mathsf{isa}(f, \mathsf{file}))$ (with f being interpreted as universally quantified). Unlike with state constraints, some violations of this condition may already exist in the initial state, but the planning agent is prohibited from creating any further violations. However, as in our work with state constraints, in order to determine whether an action is applicable in a given state, it is required that we check whether the resulting state violates any of the conditions (in addition to checking the preconditions of an action). In their formulation it is, however, required that all of the individual action descriptions explicitly enumerate changes to every predicate that is affected. State constraints of this form also appear in the original PDDL [105].

2.3.4 Interconnected physical systems

The IPC-4 version of the PSR problem, described in the axiom subsection above, is relatively simple compared to the behaviour of power networks in the real world. A more accurate model of the network requires us to compute numeric quantities such as voltages, power flows, phase angles, power generations and consumptions. While opening or closing a switch changes only the state of that switch, such an action changes the topology of the whole network and the above-mentioned numeric quantities are affected in a way that is dependent on the state of all the other switches in the network. The configuration of the network can be described only with the status of the switches (state), but these numeric quantities are higher level properties of the state (that might be of interest to us) – i.e. values of those variables are the extended state. These values depend on physical laws that can be expressed as a system of equations. Effects of actions (i.e. opening or closing a switch) on those quantities are not obvious and we are required to recompute the state of the network after each action. State constraints act as a bridge relating states of the switches (primary variables) and numeric quantities (secondary variables), as we will explain in Section 3.3.2.

The way of checking the consistency of the constraints works depends on their type. While PDDL axioms are rules only over propositional variables, our PSR example requires us to compute numeric quantities. In a power network, physical quantities (power flows, phase angles generations and consumptions) can be computed using different models with different degrees of accuracy. The AC model is the very accurate, but it involves non-linear equations and it is computationally expensive, so it is often approximated or relaxed. Other models, which use quadratic equations, include the dist-flow relaxation [5], the quadratic constraint model [28] and the quadratic approximation [100]. The linearised DC power flow model [127, 132], which we will use, lets us formulate the relevant constraints using only linear equations and inequalities. Satisfiability of the set of equations and inequalities then determines whether the configuration violates any of the constraints (i.e. whether the state is valid), so we will employ a linear programming (LP) solver for consistency checking.

While there has been some work on numeric planning (Coles et al. [30] give an overview) and on state constraints in the classical setting like axioms, work on global numerical state constraints received less attention. We will discuss two papers that describe planning in domains describing interconnected physical systems – Aylett et al. [2], who consider managing a chemical plant, Piacentini et al. [115], who address a power balancing problem in a power network. The challenge that appears in these cases is that the effects of actions are not straightforward, as we already seen above for the power network example. Similarly to us, Aylett et al. highlight the difference between well known planning domains and this type of problems: "In a robot blocks world, removing one block normally has no effect on the other blocks (as long as the blocks are taken from the top of the piles). In a process plant, the significant effect of opening or shutting a valve is not that the state of the valve changes, but that depending on the state of the rest of the plant at the time, one or more of the chemical processes might be started or stopped. The interconnectedness is reflected in the particular properties of flow." Both Aylett et al. and Piacentini et al. approached the problem by creating domain-specific planners in which a special-purpose solver is integrated with a planner.

The similarity between these approaches and the work that we will present in Chapter 3 is that the central idea is to combine classical planning with external solvers capable of reasoning about interconnected physical systems. Unlike in our work, their planners do not directly handle the discrete topological changes that affect the flow of power or of chemicals. In the case of chemical plant, opening and closing of the valves is handed over to a sub-planner, and in the case of power balancing problem, there are no switching operations.

The problem that Piacentini et al. are addressing is the electricity network balancing problem. Here the goal is to reach the end of a 24 hour period over which the zone of the power network is to be balanced – that is the elements within the network that the planner has control over have to be manipulated in such way that the demand is matched by supply at all times. Allowed actions are: (i) switching from which generators the power is generated, (ii) connecting and disconnecting different branches of the network and (iii) varying the transformer tap settings. Each of those actions has global effect. That is, they might affect the voltages and phase angles on all of the busbars and magnitude of power flowing into all of the lines.

To account for the fact that a discrete action changes the network in a non-obvuous way, the authors make the distinction between *direct* and *indirect* action effects. Direct effects are simple assignments of new values to a subset of variables. Indirect effects are not trivial to compute and have to be calculated using global constraints. Piacentini et al. integrated an existing planner, POPF [29], with an external module that solves power flow problems. POPF is a forward-chaining state-based search planner capable of dealing with numeric-temporal problems. The external solver deals with a large number of non-linear equations describing the flow of power and is used to compute indirect effects of actions. The external solver receives parametrised calls from the planner and returns the values of a subset of variables. The numeric variables in the problem are divided into *independent*, *dependent* and *special*. The variables whose values are determined by the solver are special, while the variables whose values affect the output of the solver are dependent. The variables whose values don't affect the special variables (and aren't passed to the external solver) are called independent variables. Special variables include voltages on the busbars, the power flowing into the lines and power generated by the generators. Dependent variables include settings on the transformers and the power that can be shed. (In our terminology, the independent and dependent variables are primary variables and the special variables are secondary variables. The difference is that we do not make the distinction between the variables that affect and those that do not affect the values of secondary variables.)

A similarity with our work is that they are adapting an already existing heuristic to their setting. They modified the POPF heuristic in a way that takes into account the special variables. As the effects the actions have on special variables cannot be expressed as a linear numeric function, approximations are used to determine whether an action increases, decreases or doesn't affect the value of special variable (amount by which it changes is found in the preprocessing stage).

Aylett et al. created the Chemical Engineering Planner (CEP) as a planner capable of generating plans for chemical plants. One of the requirements is that the plans must not visit any states that violate any safety rules. Their way of dealing with these global constraints is similar to the one proposed by Weld and Etzioni [140]. There are some propositions, *goals of prevention*, whose values must not be changed in any state visited by any valid plan (similar to *don't disturb* goals). When an action is added to a plan, CEP examines it to see whether it violates the constraints, and if it does, adds new preconditions to actions. Similarly to flow of power in PSR, the authors note that flow is a property of the configuration of valves in the whole network (that is, whether each of the valves is open or closed). Like Piacentini et al., Aylett et al. use an external solver, or subplanner, to calculate the flow of chemicals. This external solver is tasked with finding a flow achieving some goal. From this, it finds which valves need to be opened and which have to be closed for such flow of chemicals to occur. This is represented as a partially ordered plan and handed back to main planner which arranges the actions while respecting any constraints.

The need to model interconnected physical systems has also been discussed by Boddy and Johnson [12] in the context of *scheduling* (i.e. determining the order of actions and allocation of resources [107], rather than planning) in oil refineries and process industries. Their situation is similar to the two we described above and to the domains that we will describe in

Chaprer 3 – the problem requires accounting for the interaction between discrete choices (which in their case involves resource assignment and sequencing decisions) and continuous variables. Similar to the two above mentioned groups of authors (and similar to us), they combine a discrete solver and a continuous solver – whenever a discrete choice is made, it is required that the continuous solver (which may have to with hundreds or thousands of variables and quadratic equations and inequalities) finds whether the assignment is infeasible. (And if it is, the choices made by the scheduler need to be refined and the refined solution needs to be handed back to the quadratic solver.)

2.3.5 Hybrid systems

By hybrid planning, we mean planning in domains that involve both propositional and numeric variables.

An early example of such a hybrid planner was Zeno by Penberthy and Weld [114] which was capable of dealing with discrete finite variables, continuous resources and time. Processes were described with differential equations. Its limitation was that the concurrent actions cannot apply continuous effects on the same variable. This requirement results from the fact that the simultaneous equations must be consistent with one another, rather than accumulating additive effects. Other early approaches to hybrid planning involved compilation of problems into integer programming problems or Satisfiability Modulo Theories (SMT) instances [7]. Examples of compilation integer programming include work by Kautz and Walzer [92] and Vossen et al. [139] An of a planner that compiles problems into SMT is LPSAT by Wolfman and Weld [142], capable of solving planning problems with propositional and metric variables. The solver consists of SAT part and linear programming (LP) part (hence the name). The SAT solver activates a subset of constraints that are passed to the LP solver. If the LP solver can find a solution, it is decoded into a plan; otherwise the conflicts are added to the SAT problem in form of nogoods and new problem is handed back to the SAT solver. This is repeated until a plan is found (or the problem is determined to be unsolvable). TM-LPSAT by Shin and Davies [129] is an extension that translates the problems with durative actions and continuous change to numeric quantities to LPSAT problems and extracts the plan from the solution returned by LPSAT (therefore extending its capabilities compared to original LPSAT). Unfortunately, TM-LPSAT is reported to be very slow.

Scala et al. [126] build on the SMT approach by adding the capability to express and solve problems with *disjunctive global constraints*. Like us, they make distinction between variables directly affected by action effects and variables whose values are computed by global constraints (and each variable strictly falls into one of those two categories). One of the key differences from our work is that they allow for actions to directly modify values of both numeric and propositional variables, i.e. primary variables can be numeric. While propositional effects are state-independent, the interpretation of numeric effects is state-dependent. For example, an action can increase value of some variable by a fixed amount, so the value in the resulting state depends on the value in the state the action was applied in. This means that if such action is applied twice (or more times) in a row, its effects are different each time. One of the important features of their encoding is that it exploits this property by allowing to roll up many instances of a given action with state dependent numeric effects so they are executed in a single plan step, subject to constraints determining the validity of such sequence. This greatly reduces the planning horizon at which the theory models valid plans. The authors describe the conditions under which rolling up actions is possible, while guaranteeing that the constraints are not violated. Another important difference from our work is that the global constraints are allowed to be disjunctive[4]. For example, in the Rovers domain from the numeric track of the 3rd International Planning Competition [49], the areas constituting obstacles are modelled by as a disjunction of linear inequalities.

Like work by Scala et al., the Kongming planner by Li and Williams [102] generates hybrid plans (i.e. actions effects can modify both discrete or continuous variables) and is capable of reasoning with global constraints. The planner works by creating a *Hybrid Flow Graph* which combines planning graph for discrete actions and *flow tubes* (a way of representing infinite number of trajectories by finding the boundaries of reachable space) for continuous actions. Hybrid Flow Graph is then encoded as a mixed logic linear/non-linear program (ML(N)LP) and solved using off-the-shelf solver. The domain the Kongming planner has been applied to is guiding an autonomous underwater vehicle (AUV), and constraints are used to represent obstacles (same as in the Rovers domain).

Löhr et al. [103] combine planning techniques with methods from control theory to create what they call *Domain Predictive Control*. The control of dynamic systems aims to minimise deviations of continuous state variables from the given reference values. The behaviour of those systems is modelled by a set of linear differential equations and the goal is to generate the input

[4]In our framework presented in Chapter 3 disjunctive constraints can be modelled in a very inefficient manner. We could, for example, split each action that activates a given disjunctive constraint into as many copies as there are disjuncts. Alternatively, we could employ a solver capable of determining consistency of disjunctive constraints.

that guides the system from its initial state, described using only continuous variables, to the desired set point within the given time interval. Their work, in contrast, addresses hybrid systems which contain a number of discrete (in their case Boolean) variables that are used to model reconfigurable dynamics, like *mode switches* or logical dependencies between input signals and state transitions. They demonstrate how to generate input signals for hybrid systems, taking the system dynamics and different modes into account. The choice of the input for logic variables is, in part, done by employing a planner (they integrate Temporal Fast Downward [44]). Unfortunately, this approach is not well-suited for the problems we address as it works for modelling the association of only a small number of discrete modes. For example, in the PSR domain the number of modes required to model it using their methods would be equal to the number of possible network configurations (this is 2^n with n being the number of switches).

2.4 Relation to semantic attachments and planning modulo theories

As evident from the preceding section, certain kinds of state constraints let us combine propositional planning with non-propositional variables, such as numeric quantities. Here we will discuss two other ways of integrating non-propositional variables with planning, namely semantic attachments [40] and Planning Modulo Theories (PMT) [68].

2.4.1 Semantic attachments

Dornhege et al. [40] introduce a feature that they call *semantic attachments*, Semantic attachments allow for numeric and other types of non-propositional state variables, whose values are computed by calling a function. These functions are "external" to the planner in the sense that their meanings are not defined in the planning model (i.e. the way those values are computed is opaque to the rest of the planner). They return the new values for a given input without having any more information about the planning domain and problem than the input that they get. This allows us to integrate efficient (separately developed) tools, such as motion planners, to calculate values of a subset of variables. Semantic attachments are, therefore, a framework for integrating a planner employing well-known planning techniques such as heuristic state-space search (also called *high-level*, *symbolic* or *task* planner by the authors), with *low-level* solvers or reasoners for sub-problems.

An example from robotics is a class of problems which involve combining high-level planning (i.e. deciding on which actions to perform) with motion planning. These examples include taking the dishes out of the washer and stacking them to dry, building a model of a house or rearranging furniture in a room to a specific configuration. While symbolic planner is capable of generating a high-level plan, it is incapable of solving the sub-problems, such as determining the specific motion trajectories, which can be computed using specialised tools. The task planner is suitable for deciding on which objects to move, while the motion planner is suitable for finding a path through which to move (i.e. deciding on *how* the chosen actions are performed).

The function is invoked to provide information to the higher level planner during the planning process and only if relevant to the planner. There are two kinds of semantic attachments – *condition checkers*, which tests whether some action precondition is satisfied, and *effect applicators*, which compute changes to state variables. For example, an effect applicator for a "put down" action in a domain where a robot is manipulating objects, a low-level reasoner would be called to determine the new position of an object and update the state variables representing the position (i.e. three variables representing the position and three variables representing the orientation). There is a requirement that both the condition checker and effect applicator are deterministic. The condition checker must always terminate and return a truth value, which should always be the same for identical parameters and world states (in that sense condition checkers compute derived predicates). Effect applicators must always terminate and result, for identical parameters and states, in identical settings of the state variables they act on. The authors, however, stated that they want to remove this restriction and enable the planner to branch over an initially unknown, but finite number of outcomes.

An interesting domain illustrating the concept is an extension of the Logistics domain from International Planning Competition. The domain models a logistics problem, in which trucks are required to deliver packages from one place to another. In the original formulation, each truck can carry unlimited number of packages. This problem becomes more challenging if the trucks have defined capacities, the packages have defined sizes and we allow the trucks to carry as many packages as they can fit. Both storage spaces and the sizes of the packages are three dimensional, having height, length and width. To determine whether an arrangement of packages can fit in a truck, it is not enough to simply sum up their volumes and see whether it is less than the capacity (it is easy to see that two cube-shaped boxes of volume v would not fit into a cube shaped space of volume $2v$). Although symbolic planners cannot solve the three-dimensional packing problem, there are, however, specialised algorithms that are capable of computing this. The action of picking up a package might have a precondition with a semantic attachment that uses a solver to determine whether all of the packages can fit into the truck. The modelling of this domains has a number of parallels with our use of numeric constraints that we will describe in Chapter 3 – the specialised solver, which is unaware of the planning aspect of the problem, is used to determine the validity of a given state. In the extended logistics domain, the state is defined by the discrete variables determining which truck or city is which package in, while in the PSR domain used in Chapter 3, the discrete variables are the positions of the switches. As a side effect, the solver also computes values of the secondary variables. In this case those are the exact positions of the packages inside the truck, while in PSR those are the variables like flows of power in the lines, phase angles on the busbars and generations in the generators.

While semantic attachments are similar to our use of LP solver when dealing with numeric state constraints or Piecentini's use of solvers for power networks, there is an important difference. With state constraints, states are defined by primary discrete variables, and a solver capable of dealing with the constraints computes some properties of the state (such as values of secondary variables), but it does not change values of primary state variables. In contrast, semantic attachments allow for action applicators to manipulate state variables (whose values can still be altered by actions). Condition checking, such as determining whether the boxes can fit into the truck, however, means determining whether a set of constraints is satisfiable.

2.4.2 Planning modulo theories

Gregory et al. [68] describe PMT as an extension to planning, analogous to the way in which SMT extends the propositional satisfiability (SAT) problem. The authors observe that classical planning is similar to the SAT problem in that both deal with propositional variables. SMT is an extension of SAT in which the formulae retain their core structure, and therefore core SAT solving technology can be reused, while expressive power is increased by allowing for use of arbitrary predicate symbols, provided that there are interpretations of symbols given in a corresponding module that can be plugged into the core solver. PMT, similarly, allows for addition of new types (such as sets, vectors etc.) to the planning model and defining functions operating over those types that can be used in actions.

Unlike semantic attachments, where the extension is based on using low-level reasoners to compute functions, Gregory et al. focus on the addition of theories for types. To paraphrase, in semantic attachments, external module used takes a state as an input and returns new valuations of a subset of variables. In contrast, in PMT, actions use functions and predicates with known parameters as inputs. The type of each of the inputs is defined for each function, which makes manipulation of variables more transparent. That is, the planner knows which variables are being passed to the function and what are their types. The functions here are more transparent (compared to functions in semantic attachments), so this information can be used in computing heuristics.

PMT and planning with state constraints both allow for extending planning problem to manipulation of non-propositional variables, but they best described as orthogonal extensions. While in planning with state constraints actions directly affect values of only (primary) propositional variables and values of secondary (possibly non-propositional) variables are computed via state constraints, the PMT framework (like semantic attachments) allows for values of non-propositional variables to be directly affected by functions. PMT, however, does not have a way of dealing with global state constraints.

2.4.3 Heuristics in semantic attachments and PMT planners

Both Dornhege et al. [40] and Gregory et al. [68] implemented state-space search planners. As in our work, both groups were interested in developing delete-relaxation based heuristics to guide the search.

Dornhege et al. created extensions of the FF planner, a classical planning system based on forward state-space search [85], and Temporal Fast Downward, a planning system which extends original classical planning system Fast Downward [77] with durative actions as well as numeric and object fluents [44]. Their extension to FF planner (called FF/M) incorporates the semantic attachments in calculation of FF's delete relaxation based heuristic. The condition checkers and effect applicators, when invoked by the symbolic planner take a Boolean parameter h, which, when set to *true*, aims for faster computation, at the expense of accuracy. The symbolic planner can request this approximations when computing the heuristics. FF/M treats the conditions computed by a module the same way as the assignment of variables in the symbolic portion of the planning task – once the condition is met in a relaxed state, it remains true in all the subsequent relaxed states (therefore preserving the monotonicity). In some domains, the authors manually added domain-dependent rules for whether some condition has been achieved in a relaxed setting. For example, in the extended logistics domain, the total volume of the packages must not exceed the truck capacity, so evaluating this requirement can be used instead of calling the condition checker.

Gregory et al. explored both compilation to SMT, using a form of planning graph-based encoding, and heuristic search. Like us, they formulated an equivalent of delete relaxation and the h_{max}. Central to their approach is the concept of *domain abstraction* (this is different from *abstraction* that we talked about in Section 2.2.3). Some examples of domain abstractions given include the three-value semantics, $\{true, false, unknown\}$, for arbitrary propositional sentences, and the replacement of domains of real numbers with intervals (as it is done in heuristic computations in MetricFF [83]). Formally, domain abstraction is a *join semi-lattice*, which is a set with partial order and a defined *join* or *fold* operator. When an action is applied in a relaxed state, effects of actions, instead of assigning new values to variables combine the previously held values with the new values using this operator. Abstract domains are then used to define the relaxed state space by simply replacing the domain of each variable with its domain abstraction. As in monotonic relaxation, applying a set of actions to a relaxed state creates a new abstract state, in which the set of values for each of the variables is a superset of the set of values in which the actions were applied (i.e. the relaxation is monotonic). This lets us build the relaxed planning graph which can be used to perform relaxed reachability analysis and compute h_{max}.

The issue that remains is how to create a domain abstraction for a given variable domain. The authors provide several examples. One example of is what they call *enumerated abstraction.* Give a domain $\mathcal{D}$ it consists of a power set $\mathcal{P}(\mathcal{D})$, ordered by set inclusion and with set union as a join operator. Unfortunately, this abstraction is infinite for infinite domains. The authors discuss *bounded enumerated abstraction* and *finite basis abstraction*, which are two ways creating domain abstractions for variables with infinite domains. The finite basis abstraction simply adds a special limit value to the power set – for example, domain of integers can be abstracted as $\{\{0\}, \{1\}, \{0, 1\}, large\}$, where *large* represents any set with a value greater than 1. Bounded enumerated abstraction is a generalisation of the intervals abstraction from MetricFF and works by finding bounds on the range of possible values.

Chapter 3

Numeric state constraints

In Section 2.3, we introduced some examples of work with state constraints and discussed some of their general properties. In this chapter we will introduce a concrete type of state constraints called *numeric state constraints*, show how domains can be expressed using them, and adapt heuristics introduced in Section 2.2 to this setting.

This chapter is structured as follows. In Section 3.1 we will give an overview of the types of problems that initially motivated us to formalise and implement these types of constraints. In Section 3.2 we will introduce a formalism for numerical state constraints by extending the previously introduced FDR from Section 2.1.1. In Section 3.3 we will present a number of example domains. We will then (Section 3.4) discuss the relation to classical planning, and the complexity of problems with numeric state constraints. In Section 3.5 will deal with the adaptation of heuristics. We will conclude the chapter (Section 3.7) with a brief discussion of future work.

3.1 Motivation

We were motived by the need to apply planning techniques in environments that form interconnected physical systems (Section 2.3.4). In these domains, a planning agent manipulates the system through discrete control actions, yet the interaction between elements of the system (or subsystems) needs to be described using continuous quantities. Performing a single control action might change the state of the whole system in a way that depends on the states of all the other components. For example, closing a switch in a power network will affect the flow, not only in the given power line, but also throughout the network. Rules that govern the behaviour of continuous quantities might arise either from physical laws or from requirements of the problem and are referred to as numerical state constraints. This addition is a small step away from classical planning (as we will see in Section 3.4, the domains presented here can be compiled to classical planning, although with potentially exponential increase in problem size) and towards hybrid planning, yet it enables us to model relevant properties of various practical problems. Initially, we focused on the power network reconfiguration (Section 3.3.2) as a practical example domain. It is, however, important to emphasise that the approach is likely to be useful in dealing with various other types of practical problems, such as water and gas networks or transportation systems. In this chapter, we will only consider state constraints consisting of linear equations and inequalities, although our framework is not limited to the constraints of that type.

While numerical state constraints and metric planning both allow us to use non-discrete variables in planning, this is done in different ways. Here action effects and the initial state description assign values *only* to discrete variables. The planning agent is then free to choose any assignment of numeric variables which satisfies the constraints. This approach can be contrasted with metric planning in which values of numeric variables are directly altered by action effects, but which lack mechanisms for expressing global constraints. Metric planning and usage of global numeric constraints can therefore be thought of as orthogonal extensions to classical planning.

3.2 Formalism

Here we will define formalism for planning with numeric state constraints and explain how it relates to FDR (Definition 3) introduced in Chapter 2.

3.2.1 State

Recall that in FDR, states are assignments to a finite set of variables, each of which has a finite domain of values. Preconditions of actions and goals are then defined as partial valuations over those variables, while action effects assign new values to a subset of variables.

We enrich this formalism by adding a second set of variables: variables of the classical planning model will be referred to as primary variables, V_P, while the variables that appear in the second set are referred to as secondary variables, V_S. Unlike primary variables, secondary variables are not restricted to finite domains – when dealing with numeric state constraints, the domains of secondary variables are real numbers. The relationship between the primary and secondary variables is defined through a set of state constraints.

Because the primary variables function in the same way as state variables in FDR, terms in Definition 1 and Definition 2 apply to the primary variables. That is, partial variable assignment over the primary variables is defined the same way as the partial variable assignment over state variables in FDR.

Definition 13. *A* state *is a partial primary variable assignment s of values to all variables in the set V_P of primary state variables.*

We write $s[v]$ for the value of a primary variable v in s, and $s[p]$ to denote the value of partial primary variable assignment p in s (*true* if p holds in s, *false* otherwise).

Borrowing the terminology originally used [77] for axioms (Chapter 5), we define an extended state.

Definition 14. *An* extended state*, s_e, is a partial primary variable assignment to values to all variables in the set V_P, and an assignment to each $v_S \in V_S$ a value $s_e[v_S] \in \mathcal{D}(v_S)$ from its domain, such that all of the constraints in C_{inv} are satisfied.*

(C_{inv} will be defined below.) We will occasionally refer to state (i.e. an assignment only over primary variables) as *reduced state.* In a planning problem with global numeric constraints, there can be multiple (or potentially an infinite number of) extended states corresponding to a single reduced state.

3.2.2 Planning problem

The definition of a planning problem is modified from Definition 3 as follows:

Definition 15. *A planning problem with numeric state constraints is defined by a tuple $\Pi = \langle V_P, V_S, A, C_{\mathsf{inv}}, s_0, G, cost\rangle$ where:*

- *V_P is a set of primary variables, with each $v_P \in P_V$ being associated with a finite domain $\mathcal{D}(v_P)$.*
- *s_0 is an initial assignment of values to all variables in V_P (or initial state).*
- *V_S is a set of secondary variables. Each $v_S \in V_S$, is associated with a domain $\mathcal{D}(v_S)$ of real numbers, optionally with an upper bound, a lower bound or both upper and lower bounds.*
- *A is a set of actions, each action a defined by:*
  - *a partitioned condition* $\mathrm{pre}(a) = \langle \mathrm{pre}_P(a), \mathrm{pre}_S(a)\rangle$ *called the* precondition*, and*

   – *a partial primary variable assignment* eff(a) *called the* effect*.*

- C_{inv} *is a set of invariant switched constraints.*
- $G = \langle G_P, G_S \rangle$ *is a partitioned goal condition.*
- *cost*(a, s) *is a cost function, which is defined the same as in Definition 3. Here s can be an extended state.*

Switched constraints and partitioned conditions will be defined in the following two subsections.

3.2.3 Switched constraints

Primary and secondary variables interact only through *switched constraints*.

Definition 16. Switched constraints *are of the form $\varphi \to \gamma$, where*

- φ *is a partial primary variable assignment, which we call the* trigger*.*
- γ *is a constraint over the secondary variables. It has a form of a linear equation or a linear inequality.*

In a state in which the trigger holds, we say that the switched constraint is active*.*

It should be noted that very often we have switched conditions that should be active in every state, in which case the trigger is simply the empty partial variable assignment, $p_\emptyset$ (recall that the empty partial variable assignment holds in every state). For example, in our power network reconfiguration domain (see Section 3.3.2), these are the laws of physics, such as Kirchoff's laws.

In some of our example domains (see Section 3.3), we will use switched constraints that are active in states in which value of some primary variable is unequal to some value. While this can be expressed as a set of switched constraints with different triggers, it is more convenient to abbreviate this set of constraints as a single constraint with an inequality in the trigger. For example, if a constraint over secondary variables γ is satisfied in all states in which a primary variable v (whose domain is $\mathcal{D}(v) = \{1, ..., m\}$) is not equal to some value k (with $k \in \mathcal{D}(v)$), this means that we have $m - 1$ switched constraints of the form

$$v = i \to \gamma \qquad\qquad i = 1, \ldots, m, i \neq k$$

To abbreviate, we write this set of constraints as

$$v \neq k \rightarrow \gamma$$

Switched constraints are used in two ways – in a set of invariant constraints and as secondary part of partitioned conditions (this will be defined in Section 3.2.4). The set of invariant constraints, denoted C_{inv}, is defined in every problem and are used to determine state validity – all of the constraints in this set must be simultaneously satisfied in every state visited by a plan. As the secondary part of partitioned conditions, switched constraints appear as secondary action preconditions and secondary goals (denoted $\mathrm{pre}_S(a)$ and G_S, respectively, in the Definition 15). This means in order for an action to be applicable in some state, the union of C_{inv} and the secondary action preconditions associated with that action must be satisfied in that state. Similarly, the union of C_{inv} and the secondary goals must be satisfied in every goal state.

Formally, we define a set of active constraints and a valid state:

Definition 17. *If C is a set of switched constraints, then the set of* active constraints *(from C) in a state s is*

$$\mathrm{active}(C, s) = \{\gamma \mid \varphi \rightarrow \gamma \in C, \varphi \textit{ holds in } s\}.$$

Given a planning problem, Π *with a defined set of invariant constraints C_{inv}, a state s is* valid *iff* $\mathrm{active}(C_{\mathsf{inv}}, s)$ *is satisfiable.*

It should be noted that $\mathrm{active}(C_{\mathsf{inv}}, s)$ is a collection consisting only of right-hand sides of switched constraints. Consequently, a solver that we use to check satisfiability does not need to deal with the primary variables. All that is required is a solver that is capable of dealing with the type of constraints on secondary variables. As in planning with numerical state constraints the domains of secondary variables are real numbers, and constraints over secondary variables are linear equations and linear inequalities, satisfiability of $\mathrm{active}(C_{\mathsf{inv}}, s)$ can be checked using a linear programming solver.

An interesting thing to note is that switched constraints can be used to encode conditions on primary variables. For example, if $p_1, \ldots, p_k$ are primary variables with Boolean domains, and C_{inv} includes the constraints $(p_i = \mathit{true}) \rightarrow (v_i = 1)$ and $(p_i = \mathit{false}) \rightarrow (v_i = 0)$ for $1 \leq i \leq k$, then $G_S = \{\mathit{true} \rightarrow (\sum_{i=1}^{k} v_i) = 1\}$ enforces that exactly one of $p_1, \ldots, p_k$ is true in any goal state. The concept of expressing conditions on primary variables in this way will be elaborated on in Section 3.4.

Other types of constraints

While the secondary variables discussed in this chapter are real numbers and the constraints over the secondary variables we will focus on are linear equations and inequalities, it is important to note that the formalism discussed in this chapter (and the heuristics that we will develop) remains valid for other types of constraints. I.e. we can retain the Definition 15, but change the definition of secondary variables V_S and/or the form of the constraint over secondary variables (switched constraints retain the same form $\varphi \rightarrow \gamma$). The only requirement is that removing a constraint from a consistent set of constraints (i.e. making a switched constraint inactive) will not make the set inconsistent. That is, we require that if a set of constraints is satisfiable, all of the subsets are satisfiable as well. While this requirement is true for most systems of constraints, there are counter examples, like non-monotonic logics such as default logics [119].

3.2.4 Partitioned condition

Definition 18. *A* partitioned condition *is a pair $\langle c_P, c_S \rangle$, where c_P is a partial primary variable assignment and c_S a set of switched constraints. We will refer to c_P as the primary part and c_S as a secondary part of a partitioned condition.* $\langle c_P, c_S \rangle$ holds *in state s iff*

- $s[c_P]$ *holds and*
- $\text{active}(c_S \cup C_{\mathsf{inv}}, s)$ *is satisfiable.*

A partitioned condition with an empty partial variable assignment in the primary part and an empty set of switched constraints in the secondary part holds in every state. We will denote such partitioned condition $\varphi_\emptyset$.

As stated in Definition 15, action preconditions and the goal are partitioned conditions. We will call the first and the second part of partitioned conditions the primary and secondary precondition and primary and secondary goal, respectively. E.g. in our power network reconfiguration example, positions of the switches in a network are primary variables, while whether the given users are supplied with power are secondary variables. A primary goal might consist of specifying positions for a subset of switches and a secondary goal might specify which of the buses are supplied with power. In every goal state, the configuration of switches specified in the primary goal is true and the union of invariant constraints and goal constraints is satisfiable.

3.2.5 Actions

Action effects function the same way as in FDR (Section 2.1.1) – applying an action assigns new values to a subset of primary variables. Formally, application of a in s changes the values of every $v \in \mathcal{V}(\text{eff}(a))$ to $\text{eff}(a)[v]$ and we denote the resulting state by $s[\![a]\!]$ (values of all the other variables remain the same as in s). We limit the action effects to this form as we wanted to keep the state space finite, in line with Assumption 1 of classical planning (Section 2.1).[1]

Determining whether we are allowed to apply a given action in a given state is, however, more complicated than in classical planning, as we make distinction between whether the action is *applicable* and whether the action is *allowed* in that state. While in classical planning an action precondition is a partial variable assignment (Definition 3), in planning with numeric state constraints, an action precondition is a partitioned condition. We still say that an action a is *applicable* in state s iff its precondition, $\text{pre}(a) = \langle \text{pre}_P(a), \text{pre}_S(a) \rangle$, holds in s (see Definition 18). However, for an action to be *allowed* in s, it has to be both applicable in s and the resulting state $s[\![a]\!]$ has to be valid. We can think of this as an implicit precondition.

In our PSR example actions are opening and closing of the switches. Phase angles are secondary variables with a range between $-\pi$ and π and we are not allowed to close a switch if the difference between the phase angles on two sides of a switch is above some threshold value. As an example suppose that we have a switch s that is associated with a power line that connects two buses b_0 and b_1 (which are associated with phase angles θ_0 and θ_1). Also suppose that we want to make sure that the difference between the phase angles in the buses that are to be connected below θ_t. In that case, the precondition of an action $\mathsf{close}(s)$ is $\langle s_{\text{closed}} = \mathit{false}, \mathit{true} \rightarrow \mid \theta_0 - \theta_1 \mid \leq \theta_t \rangle$. That is, for the action to be applicable both the switch must not be closed and the union of secondary precondition and the invariant constraints must be satisfiable. Capacity constraints and Kirchoff's laws are examples of invariant constraints in the PSR domain – for the action action to be allowed (in addition to being applicable), the resulting state must obey those constraints.

We should note here that introducing the concept of allowed actions is not the only way to model our domains. We can think of a portion of pre-

[1]If this assumption was removed, we could allow for the planner to have more control over the secondary variables, for example, via numeric parameters to actions. This would place us, however in the domain of metric and hybrid planning (Section 2.3.5), and the techniques that are applicable would have to be very different. Keeping as close as possible to the classical planning model enables us to adapt techniques that were developed in this area, while still tackling problems that would be difficult to model and solve using purely classical formulation.

conditions as being hidden in the secondary model – i.e. it can be argued that we are actually working with incomplete models of actions. While introducing the concept of allowed actions could be avoided by placing more information in action preconditions or in goals, the differentiation between valid and invalid states (and therefore allowed and non-allowed actions) more clearly highlights what situations we want to avoid. For example, we could model the PSR domain in which we can always close or open any switch, but this action may result in component failures. This could than be addressed by having *no component failures* as part of the goal. However, we found that modelling the actions with implicit preconditions is more straightforward.

3.2.6 Plans

As in classical planning, $s[\![\langle a_1, ..., a_k \rangle]\!]$ denotes the state that is obtained by sequentially applying the actions $a_1, ..., a_k$, starting from state s. The only difference is that we require all the actions to be allowed in the state in which they are applied (that is, all the states in a sequence induced by the sequence of actions are valid). The definition of an s-plan is analogous to the Definition 4 given in Section 2.1.1.

Definition 19. *Given a problem* $\Pi = \langle V_P, V_S, A, C_{\mathsf{inv}}, s_0, G, cost \rangle$ *and a state s, a sequence of actions $a_1, ..., a_k$ is called an s-*plan *if G holds in* $s[\![\langle a_1, ..., a_k \rangle]\!]$.

We have previously defined (Definition 3 in Section 2.1.1) the cost of an action a as a function of the action and the state s in which a is applied – $cost(a, s)$. The definition of an optimal plan (Section 2.1.1) remains unchanged: It is a plan whose cost is minimal among all plans starting from an initial state.

Note that when the domain involves secondary variables, the cost of an action can be a function of an extended state. This can present an issue if there are multiple (or an infinite number of) extended states corresponding to a single reduced state. If that's the case, how do we determine the values of secondary variables that the action cost depends on?

In this chapter, we only deal with domains where action costs are not state dependent and the heuristics that we will present in Section 3.5 were developed under this assumption. Chapter 4, which will discuss planning with state-dependent action costs in more detail, will focus (mostly) on the case where all the values affecting action costs are uniquely determined given an assignment of primary variables and, consequently, the above question can be avoided.

In the preceding subsection (3.2.5), we discussed the reasons for not allowing the planner to directly control the secondary variables through action parameters. However, in domains where are multiple (or infinite number of) extended states corresponding to a single state s and where those states have different costs, the planner may be allowed to minimise the cost by picking the values of secondary variables such that the cost function is at minimum. This is still consistent with the restrictions on action effects explained in Section 3.2.5. Allowing this kind of manipulation of secondary variables may or may not be reasonable depending on the problem that we are trying to model.

3.3 Domain examples

To illustrate the power, and limitations, of the formalism, we present four examples of domain models. We also note that the way the problem is modelled might influence the accuracy of problem's relaxation – we will provide an example of this in Section 3.5. As a notational convention throughout this work, constants are distinguished from variables by an overline bar ($\bar{c}$ vs v).

3.3.1 Hydraulic blocks world

This domain is an extension of the *Blocks World* domain. The Blocks World was originally developed by Winograd [141] as a test bed for his program for understanding the natural language, but it was subsequently used much more widely as a test bed for planning algorithms [107]. The Blocks World consists of a finite number of blocks stacked into towers on a surface large enough to hold all of the blocks. The positioning of the towers on the surface is irrelevant (i.e. state is defined by which block is in which tower and by positions of the blocks within the towers). The planning problem is to turn the initial state into a goal state by moving one block at a time from the top of a tower (i.e. it is only possible to pick up a block if it is on the top of the tower) onto the top of another tower or to the table [130].

Our extension of the problem is called *Hydraulic Blocks World* (HBW). As before, we have a fixed number of $\overline{m}$ towers and $\overline{n}$ blocks. The differences are that each of the blocks has an assigned weight and that each tower k sits on a piston inside a vertical cylinder with area $\overline{a}_k$, rising from a sealed reservoir of hydraulic fluid, as illustrated in Figure 3.1(a). Each block i has a weight $\overline{w}_i$. The height of each piston is determined by the total weight of the blocks in each cylinder and their areas, observing the physical law

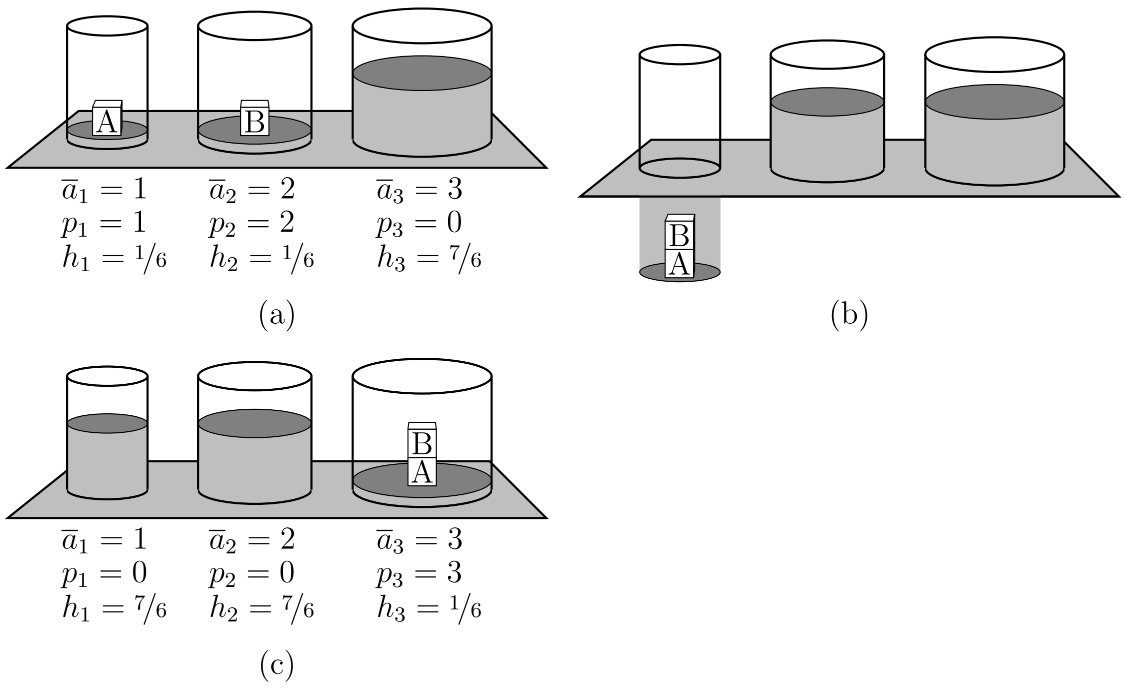


Figure 3.1: An example of the Hydraulic Blocks World domain. (a) A valid (initial) state: The weight of block A is 1 and the weight of block B, which sits on a twice as large area, is 2, causing pistons 1 and 2 to balance at the same height while the empty piston 3 rises higher. The total volume of fluid is $\bar{v} = 4$. (b) An invalid state: Placing B on A in cylinder 1, the combined weight causes the piston to fall through the bottom of the cylinder. (c) A valid goal state: Placing the total weight of A and B on the larger piston 3 makes it possible to counterbalance the weight with smaller fluid columns in the other cylinders.

stating that the pressure that each column exerts on the reservoir must be equal and that the total volume of fluid remains constant (the fluid is non-compressible). As in original Blocks World, the goal is to rearrange the blocks until the goal configuration of blocks is reached. However, we also have an additional requirement that the fluid in each of the cylinders must not fall below some minimum level or rise above some maximum level.

Actions are the usual pickup, $\mathsf{putdown}$, $\mathsf{unstack}$ and stack, indexed to indicate what cylinder the block is moved to or from. For example, $\mathsf{unstack}_{i,j,k}$ takes block i off block j in cylinder k.

Primary state variables are pos_i, representing the position of block i, whose domain is the set of pistons, other blocks and the constant $\mathsf{in\text{-}hand}$; in_i, representing the which cylinder block i is in, with domain $\{1, \ldots, m\} \cup \{\mathsf{none}\}$; a Boolean variable clear_i, representing if block i is clear, and $\mathsf{holding}$, whose domain is the set of blocks and none. The preconditions and effects of actions on the primary variables are as expected. For example, $\mathsf{unstack}_{i,j,k}$ requires $\mathsf{pos}_i = j$, $\mathsf{in}_i = k$, $\mathsf{clear}_i = \mathit{true}$ and $\mathsf{holding} = \mathsf{none}$, and causes $\mathsf{pos}_i = \mathsf{in\text{-}hand}$, $\mathsf{holding} = i$, $\mathsf{in}_i = \mathsf{none}$, $\mathsf{clear}_i = \mathit{false}$ and $\mathsf{clear}_j = \mathit{true}$. Actions have no secondary preconditions.

The key secondary variable is the height h_k of the fluid column in each cylinder k, and the main safety constraint is that this variable remains above 0 and below the height $\bar{l}_k$ of the cylinder:

$$0 \leq h_k \leq \bar{l}_k \tag{HBW.a}$$

for $k = 1, \ldots, m$. (Note that this is actually a switched constraint, whose triggering condition is $p_\emptyset$. We omit the trigger for such constraints to simplify the notation.) The total weight of the tower of blocks in cylinder k is represented by a secondary variable p_k. To compute p_k, we use secondary variables $p_{i,k}$, $i = 0, \ldots, n$, $k = 1, \ldots, m$, representing the contribution that block i makes to the total weight in cylinder k. $p_{i,k}$ is either 0, if block i is not in cylinder k, or the weight of the the block, $\overline{w}_i$, if it is. This is enforced by the following switched constraints:

$$\mathsf{in}_i \neq k \rightarrow p_{i,k} = 0 \tag{HBW.b.i}$$
$$\mathsf{in}_i = k \rightarrow p_{i,k} = \bar{w}_i \tag{HBW.b.ii}$$
$$p_k = \sum_{i=1}^{n} p_{i,k} \tag{HBW.b.iii}$$
$$0 \leq p_{i,k} \leq \bar{w}_i \qquad i = 1, \ldots, n, k = 1, \ldots, m \tag{HBW.b.iv}$$

Constraint (HBW.b.iv) is redundant, since it is implied by (HBW.b.i–HBW.b.ii). However, as we will see in the section on relaxations, adding redundant constraints to the secondary model can improve the inference power of relaxations. We can determine h_k via the following system of equations, which state that (HBW.c) the total amount of fluid $\bar{v}$ in the cylinders is constant, (HBW.d) the force f_k at the bottom of cylinder k is proportional to the weight p_k of the tower of blocks plus the weight of the fluid column in the cylinder (the fluid density $\bar{\rho}$ times the fluid's volume, where $\bar{a}_k$ is the cylinder's cross-sectional area), and (HBW.e) the pressure (force per unit area) at the bottom of a cylinder is the same for each cylinder:

$$\sum_{k=1}^{m} \bar{a}_k h_k = \bar{v} \qquad k = 1, \ldots, m \qquad \text{(HBW.c)}$$

$$f_k = p_k + \bar{\rho}\bar{a}_k h_k \qquad k = 1, \ldots, m \qquad \text{(HBW.d)}$$

$$\frac{f_k}{\bar{a}_k} = \frac{f_{k+1}}{\bar{a}_{k+1}} \qquad k = 1, \ldots, m-1 \qquad \text{(HBW.e)}$$

Suppose the initial state is as shown in Figure 3.1(a), and that the goal is to place block B on block A. Considering only the primary part of the model, this can be achieved by picking up B and stacking it on A. However, the resulting state is not valid, because the combined weight of A and B placed on the small area in cylinder 1 exerts too much pressure; to counterbalance it, the columns in cylinders 2 and 3 would need more fluid than the total volume, $\bar{v} = 4$. A valid goal state is shown in Figure 3.1(c). Here, the weight of the tower is placed in cylinder 3, which has a larger area $\bar{a}_3 = 3$, making it possible to counterbalance the weight with lower columns in cylinders 1 and 2. This state is reachable by moving A from piston 1 to piston 3, then moving B onto A. All intermediate states in this plan are valid.

3.3.2 Switching Problems in Power Networks

Our second domain exemplifies the kind of useful problem that our approach enables planning to address. Here our task is to reconfigure a power network, by opening and closing switches that, respectively, isolate or connect power lines in the network. This can be for several purposes – in Section 2.3 we discussed the Power Supply Restoration (PSR) benchmark used in IPC-4 [43]. For another example, we may want to isolate a particular line or generator, that is being phased out for servicing, while maintaining supply to all loads at every intermediate state of the plan.

In contrast to the version of PSR used in the IPC benchmark, numeric state constraints allow us to model power flows and capacity constraints, which is an essential requirement to make the model realistic. The PSR problem consists of determining a sequence of switching operations that reconfigures a faulty network such that the faults are isolated from the rest of the system and as many customers are resupplied as quickly as possible. Faults are particularly common in bad weather conditions and faults on multiple network elements are not rare. When faults occur, circuit breakers open to protect the network from overloading, leaving the entire area connected to that circuit breaker without power. In these situations, faulty elements must be located, which is a problem that we assume is solved. A subset of switches is operated remotely, while the rest require manual operation. *Automated PSR*, which is the problem considered here and in the paper by Thiébaux et al. [134], consists of only operating the remotely controlled switches. Automated PSR must be completed within a time bound (1-5 minutes), otherwise it results in heavy fines from the power regulators. Currently, PSR is most commonly performed by human operators, sometimes aided by rule-based algorithms capable of issuing switching recommendations in very simple fault situations. The existing literature mostly deals with finding the final configuration of the network and in many cases, the problem is limited to finding a configuration resupplying areas downstream of faults assumed to be already isolated. Additionally, the existing approaches produce suboptimal solutions or make various simplifying assumptions such as assuming that all non-faulty lines can be supplied or limiting the problem to only one fault [134].

Here we formulate the PSR as a planning problem. It has been previously modelled as a mixed-integer programming (MIP) problem by Thébaux et al. [134]. Their approach is to decompose the problem into two MIP subproblems – first, the problem of finding the optimal configuration of the network is solved, ignoring the intermediate plan steps. Secondly, they solve the sequencing problem, i.e. deciding on how to drive the network into the previously determined optimal configuration. This may produce suboptimal plans, but they show that their solutions are nearly indistinguishable from optimal. Unfortunately, while MIP works for generating short plans with known lengths, it is not well suited for generating plans consisting of many actions. Our work aims to develop an alternative to MIP in such circumstances.

Power networks

The power network is modelled as a graph $\langle \mathcal{B}, \mathcal{L} \rangle$ whose nodes are buses $i \in \mathcal{B}$ supporting constant consumer loads $\bar{l}_i$; a subset of buses ($\mathcal{G}$) supply variable generation g_i (generators have capacities that constrain the power $\bar{g}_{i,max}$ produced). In the PSR problem, we also distinguish a subset $\mathcal{F}$ of faulty buses. Edges $(i,j) \in \mathcal{L}, i < j$ of the graph are power lines of the network. A subset of these edges is associated with remotely-operated switches (with one switch associated with each edge) that, when flipped, disconnect the associated line (or remove the edge from the graph).

Distribution networks have meshed topology, which is often configured radially. The switches are set so that a path taken by power from each circuit-breaker forms a distinct tree called a feeder, each element being fed by at most one circuit breaker. However, the advent of the distributed generation is gradually turning the distribution system into meshed networks for which the radiality assumption does not hold [134].

A bus is supplied with power, or *fed* (f_i), iff there is a path of lines (with corresponding switches closed) connecting it to a generator bus (in $\mathcal{G}$), in which case its entire load must be supplied. In our planning domain, this is modelled by a secondary variable f_i for each bus $i \in \mathcal{B}$. Lines also have capacities $\overline{p}_{ij}$ that limit the maximum power that can flow through them.

In this model, the only primary variables are the switch positions y_{ij} on the power lines. Opening/closing a switch toggles y_{ij} between *false* (open) and *true* (closed). The planner also varies each generator's output. This is, however, modelled by allowing the planner to assign value of the secondary variable representing generation g_i, which represent the power produced at generator bus $i \in \mathcal{G}$, rather than explicit actions. In a meshed network configuration there can be multiple solutions to the active state constraints.

As positions of the switches are the only primary variables in this domain, configuration of their positions determines the state. An interesting thing to note is that, given the configuration of the switches, the "fed" values of the buses are uniquely determined by the switched constraints (PSR.d) and (PSR.e). In Chapter 4 we will explain how we exploited this property of the domain for planning with state-dependent action costs.

The behaviour of the power grid can be modelled in several ways, so the form the secondary model takes depends on how do we decide to describe the power flow. The most accurate model is the AC model, but it is also very computationally expensive, as it requires dealing with sets of non-linear equations. For this reason, various (computationally cheaper) simplifications have been been developed. Here we use the linear DC power flow model [127, 132].

In this model, the DC power flows are a linear approximation derived from the AC power flows through a series of approximations justified by operational considerations. For the explanation of assumptions under which this model holds and the derivation, see Powell [118] and Thiébaux et al. [134]. Under these assumptions, flow of power through a power line p_{ij} between the buses i and j, $i,j \in \mathcal{B}$, is proportional to the difference between the *phase angles* associated with the buses the line connects and the *line susceptance* $\bar{b}_{ij}$ (which is a constant associated with each line). That is, $p_{ij} = -\bar{b}(\theta_i - \theta_j)$.

The DC powerflow model uses only linear constraints, so the formulation of the problem is consistent with the Definition 15. The AC model uses equations containing trigonometric functions [134][2]. This model is, however, still consistent with the requirements we outlined in Section 3.2.3, so solving power network reconfiguration problem with a more accurate power flow model would simply involve replacing the equations in the secondary model.

Adopting the DC power flow model, our secondary variables are:

- phase angles, θ_i for each $i \in \mathcal{B}$, which are real number ranging between $-\pi/2$ and $\pi/2$
- powerflows, p_{ij} for each line $(i,j) \in \mathcal{L}$, which are real numbers
- power generated, g_i for each $i \in \mathcal{G}$, which are real numbers, ranging between 0 and $\bar{g}_{i,max}$ and
- "feds", f_i for each $i \in \mathcal{B}$, which take value of either 0 or 1.

There are three main types of invariant constraints in this domain [134]. The first group of invariant constraints define the line power flows. The flow of power through a power line is given by the linearised DC powerflow model. Powerflow in the lines for which the switch is open is zero. Therefore the switched constraints are:

$$y_{ij} = \mathit{true} \rightarrow p_{ij} = -\bar{b}_{ij}(\theta_i - \theta_j) \qquad (i,j) \in \mathcal{L} \qquad \text{(PSR.a.i)}$$
$$y_{ij} = \mathit{false} \rightarrow p_{ij} = 0 \qquad (i,j) \in \mathcal{L} \qquad \text{(PSR.a.ii)}$$

Constraints of the second type govern the flow of power. These include:

[2]The AC model requires to consider both the real power flow p_{ij} and the reactive power flow q_{ij} for each line $(i,j) \in \mathcal{L}$. In addition, with constants known as susceptance $\bar{b}_{ij}$ and conductance $\bar{c}_{ij}$ are specified for each line. Variables associated with each bus i are voltage magnitude $\mid V_i \mid$ and phase angle θ_i. The AC power flow equations for a line (i,j) are:

$$p_{ij} = \mid V_i \mid^2 \bar{c}_{ij} - \mid V_i \mid\mid V_j \mid (\bar{c}_{ij}\cos(\theta_i - \theta_j) + \bar{b}_{ij}\sin(\theta_i - \theta_j))$$
$$q_{ij} = - \mid V_i \mid^2 \bar{b}_{ij} - \mid V_i \mid\mid V_j \mid (\bar{c}_{ij}\sin(\theta_i - \theta_j) - \bar{b}_{ij}\cos(\theta_i - \theta_j))$$

- Kirchoff's law of power flow. Makes sure that the power flow is conserved at the buses.

$$g_i + \sum_{j:(j,i)\in\mathcal{L}} p_{ji} = \bar{l}_i f_i + \sum_{j:(i,j)\in\mathcal{L}} p_{ij} \qquad i \in \mathcal{B} \tag{PSR.b}$$

- No faulty buses can be supplied with power.

$$f_i = 0 \qquad i \in \mathcal{F} \tag{PSR.c}$$

- All non-faulty generator buses are fed.

$$f_i = 1 \qquad i \in \mathcal{G} \setminus \mathcal{F} \tag{PSR.d}$$

- Buses that are connected must always have the same fed status.

$$y_{ij} = true \rightarrow f_i = f_j \qquad (i,j) \in \mathcal{L} \tag{PSR.e}$$

Finally, we have capacity constraints:

- Capacities of the generators.

$$0 \leq g_i \leq \bar{g}_{i,max} \qquad i \in \mathcal{G} \tag{PSR.f}$$

- Capacities on the power flow.

$$-\bar{p}_{ij} \leq p_{ij} \leq \bar{p}_{ij} \qquad (i,j) \in \mathcal{L} \tag{PSR.g}$$

Opening or closing a switch may cause transient phenomena that can threaten network stability. Whilst here actions have no secondary preconditions, using these would be beneficial to prevent certain transient phenomena which could cause instability. For example, before connecting two buses by closing a line, the difference between the phase angles on the buses $\theta_i - \theta_j$ should be relatively small. If it is not, a generation redispach is necessary to reduce it [76]. This can be accounted for by adding a secondary precondition to the closing action associated with a line (i, j) constraining the difference $\theta_i - \theta_j$ to lie within safety bounds. (This, has, however, not been used in our experiments.)

Plans

As noted earlier, reconfiguring the power network can be done for a number of possible reasons, including power supply restoration, load rebalancing, isolating a network element (so that it can be repaired or replaced) etc. A solution of a PSR problem is a sequence of switching operations such that in the final configuration a predetermined subset of buses is supplied with power. That is, the (secondary) goal is $f_i = 1$ for $i \in \mathcal{B}_{\text{goal}}$. (In our experiments, a set of buses $\mathcal{B}_{\text{goal}}$ is precomputed for each problem.)

There are different ways that the cost of a solution can be evaluated. In simplest case, we want to minimise the number of switching operations – i.e. a shorter solution is preferable as it resupplies the network sooner. A more meaningful objective is to resupply as much of the network as soon as possible – if we plot the portion of the network supplied with power as a function of time, we want to maximise the area under the curve [134]. Another objective is to minimise the deviation from standard (pre-default) configuration. Both objectives can be formulated as state-dependent action costs (see Chapter 4). However, minimising the number of switching operations is a reasonable proxy (at least for the latter objective) and is much easier for planners to do. The reason for this assumption is that we are assuming that we don't need to flip any single switch more than once. That is, if we measure the difference between the configurations by counting the number of switches that are in a different position (rather than, say, using quantities such as flows of power in different power lines), minimising the number of switching operations also minimises the difference between the initial and final configurations.

3.3.3 Multi-commodity Long-haul Transportation

Logistics problems have long been a staple planning benchmark. Our third example domain models a real-world multi-commodity long-haul transportation problem [94].

Goods, of different types, needs to be transported from a depot to customer locations, $1, \ldots, \bar{m}$. As a convention, we label the depot location 0. D_{ij}, $i, j \in \{0, \ldots, \bar{m}\}$ is the distance, along the road network, between locations i and j. Each customer i has a demand $\bar{q}_i^g$ for good type g. Transportation is done with a fleet of $\bar{n}$ trucks. Each truck k has a set $\bar{\mathcal{G}}_k$ of goods types that it can carry, a capacity $\bar{p}_k$, and a per-kilometre cost $\bar{c}_k$. In the problems we encounter, there are usually several trucks of the same type, i.e., with identical parameters. Also, there are only two goods types: ambient and chilled. Refrigerated trucks can carry both types, while non-refrigerated trucks can only carry ambient temperature goods. All trucks start at the depot and must return to the depot at the end of the plan, as well as meet all customer demands. We assume that trucks get loaded with goods in the morning and do one trip a day – i.e. they do not revisit the depot once loaded. This problem is based on the requirements a Queensland-based transportation company, which provided the data [94].

In our model of this problem, all reasoning about goods delivery is done in the secondary model. Primary state variables are loc_k for each truck k, with domain $\{0, \ldots, \bar{m}\}$, representing the current location of the truck. In addition, a Boolean variable $\mathsf{visited}_{k,i}$ keeps track of whether truck k has visited location i. The only action is $\mathsf{drive}_{k,i,j}$, with precondition $\mathsf{loc}_k = i$, effect $\mathsf{loc}_k = j$ and $\mathsf{visited}_{k,j} = \mathit{true}$, and cost $\bar{c}_k \cdot D_{i,j}$.

For each truck k, customer location i, and goods type $g \in \{\mathrm{am}, \mathrm{ch}\}$, a secondary variable $d_{k,i}^g$ represents the amount of goods type g that truck k delivers to customer i. The following constraints ensure that trucks deliver only to locations that they visit, and that type and capacity restrictions are met:

$$\mathsf{visited}_{k,i} = \mathit{false} \rightarrow d_{k,i}^g = 0 \qquad k = 1, \ldots, \bar{n}, i = 1, \ldots, \bar{m}, \qquad \text{(LH.a)}$$
$$g \in \{\mathrm{am}, \mathrm{ch}\}$$

$$\left(\sum_{i=1,\ldots,\bar{m}, g \in \{\mathrm{am},\mathrm{ch}\}} d_{k,i}^g \right) \leq \bar{p}_k \quad k = 1, \ldots, \bar{n} \qquad \text{(LH.b)}$$

The goal of meeting customer demands is expressed by a secondary goal constraint:

$$\left(\sum_{k: g \in \bar{\mathcal{G}}_k} d^g_{k,i} \right) = q^g_i \quad i = 1, \ldots, \bar{m}, g \in \{\mathrm{am}, \mathrm{ch}\} \qquad (\text{LH.c})$$

Demand and capacity values are integer, and for every problem instance there is a finite maximum. Hence, this problem can also be modelled as a classical planning problem, using only finite-domain variables.

Because our approach to optimally solving planning problems with global numeric state constraints is based on admissible heuristic search, it is affected by problem symmetries, such as those caused by identical trucks in the long-haul transportation problem. We can eliminate some of these symmetries (though by no means all) by a small reformulation of the primary model. Similarly, we can improve the accuracy of the relaxation from which derive admissible heuristics by some reformulation of the model.

First, we split the representation of the depot into two locations: the source, 0, from which all trucks depart, and the sink $\bar{m}+1$, to which they must return. drive actions are modified so that trucks can not leave the sink location or move back to the source once they have left it. To permit the plan to not use a particular truck, a zero-cost action $\mathsf{drive}_{k,0,\bar{m}+1}$ is added for each truck. Since movements of different trucks are completely independent, they can be ordered in any way. To avoid the factorial number of equivalent plans that differ only by reordering of independent actions, we force trucks to move in sequence: truck $k+1$ can only leave the source (depot) after truck k has reached the sink (depot). Furthermore, if trucks $k+1$ and k are of the same type (i.e., $\bar{\mathcal{G}}_k = \bar{\mathcal{G}}_{k+1}$, $\bar{p}_k = \bar{p}_{k+1}$ and $\bar{c}_k = \bar{c}_{k+1}$), then truck $k+1$ must be used (i.e., visit at least one customer) if truck k was. This avoids the exponential number of equivalent plans that differ only by which subset of identical trucks is used.

3.3.4 The Counters Domain

The Counters domain was invented by Francés and Geffner [51] as a way to illustrate one flaw of heuristics based on monotonic relaxation. We already discussed this domain in Section 2.3.2. Here we will show how it can be modelled with switched constraints. The original domain also featured conditional effects (the inc action has assigns different values to variables in different states), which we remove, as they are not allowed in our formalism.

As already stated, the domain features $\bar{n}$ counters, $X_1, \ldots, X_{\bar{n}}$, each ranging over integers $0, \ldots, \bar{m}$. Actions $\mathsf{inc}(i)$ and $\mathsf{dec}(i)$ increment and decrement, respectively, counter i by 1. Initial values of the counters can be all zero, all maximum, or random. The goal is $X_i < X_{i+1}$ for $i \in [0, \bar{n} - 1]$.

In our model, primary state variables are $\bar{m}$ propositional variables, $p_{i,j}$, for each counter i. The model represents $X_i = k$ with $p_{i,j} = \mathit{true}$ for $j = 1, \ldots, k$ and $p_{i,j} = \mathit{false}$ for $j = k + 1, \ldots, \bar{m}$. Formulating the actions to maintain this representation is straightforward. Because we do not use conditional effects we need a separate action for each counter value. For example, action $\mathsf{inc}(i, j)$ (with $j > 0$), which increments counter i from $j - 1$ to j, has primary precondition $p_{i,j-1} = \mathit{true}$ (except if $j = 1$) and $p_{i,j} = \mathit{false}$, and effect $p_{i,j} = \mathit{true}$. Each counter is also represented by a secondary variable, x_i. The following invariant constraints ensure that the primary and secondary representations agree:

$$p_{i,j} = \mathit{true} \rightarrow x_i \geq j \qquad i = 1, \ldots, \bar{n}, j = 1, \ldots, \bar{m} \qquad (\textsc{counters.i})$$

$$p_{i,j} = \mathit{false} \rightarrow x_i \leq j - 1 \qquad (\textsc{counters.ii})$$

$$0 \leq x_i \leq \bar{m} \qquad i = 1, \ldots, \bar{n} \qquad (\textsc{counters.iii})$$

For example, if $p_{i,1} = \mathit{true}$ and $p_{i,2} = \mathit{false}$, (COUNTERS.i) and (COUNTERS.ii) force $1 \leq x_i \leq 2 - 1$, i.e., $x_i = 1$. Constraint (COUNTERS.iii) ensures that $x_i = 0$ when $p_{i,1} = \mathit{false}$ and that $x_i = \bar{m}$ when $p_{i,\bar{m}} = \mathit{true}$.

The reader may wonder why we adopt such a complex representation, instead of simply a single primary variable with domain $\{0, \ldots, \bar{m}\}$ for each counter. The reason for this is that this is an artificial domain, modelled this way to allow for stronger relaxation and therefore better heuristic values. This will be detailed on in Section 3.5.1.

The goal is expressed on the secondary variables. To account for the fact that counter values are integer, we write the subgoals as $x_i + 1 \leq x_{i+1}$. Alternatively, we could write $x_i < x_{i+1}$ and add the requirement that the x_i variables are integer, but this would require the use of a mixed-integer programming (MIP) solver rather than just an LP solver, for checking consistency of the secondary constraints.

3.4 Expressivity

In Section 2.3.1 we mentioned that axioms can be complied away, although this leads to either a worst-case exponential blow-up in the size of domain description or worst-case exponential blow up in the size of the length of the shortest plan [136]. The question that arises is whether the formalism with state constraints is more expressive than classical planning, and what's the complexity of solving problems expressed in it.

Our first observation is that the secondary part of any partitioned condition can be compiled into the primary part. The proofs of the two propositions are presented in the appendix and will also be given in our JAIR paper [73].

Proposition 1. *Let $\langle \varphi_P, \varphi_S \rangle$ be a partitioned condition. There is a formula* $\mathrm{F}(\varphi_S)$ *over V_P such that for every state s, $s[\varphi_P \wedge \mathrm{F}(\varphi_S)] = \mathit{true}$ if and only if $\langle \varphi_P, \varphi_S \rangle$ holds in s.*

Thus, secondary conditions can be eliminated from action preconditions and the goal.

We now discuss compilation of invariant constraints. A formula $\mathrm{F}(C_{\mathsf{inv}})$ that characterises valid states in terms of the primary variables only can be constructed as in Proposition 1. Adding $\mathrm{F}(C_{\mathsf{inv}})$ to all action preconditions and to the goal ensures that a plan visits only valid states: Each state except the last must be valid for the next action to be applicable, and the final state must be valid to satisfy the goal. It follows that our formalism can be reduced to classical planning, though potentially at the expense of an exponential increase in problem size.

Is the blow-up in the size of the problem avoidable? This depends on the the kind of global state constraints in use. However, our second proposition, which will be given below provides a partial answer. Recall that although in this work we focus on switched constraints over real-valued secondary variables, our approach to extending classical planning with global state constraints, as defined in Section 3.2.3, is independent of the constraint language. Proposition 1 holds no matter what kinds of secondary variables and constraints over them appear in the problem. This also implies that, independently of the size of the problem that results from compiling away global state constraints, we cannot say anything about the time complexity of performing the compilation, since checking if a set of constraints are satisfiable in a given state may not even be decidable.

Our second observation is that we can encode complex conditions over the primary variables into secondary constraints, provided the constraint language is sufficiently expressive. In particular, within the formalism of linear switched constraints over real-valued secondary variables we can formulate action preconditions and goals that are equivalent to general formulas over the primary variables. Therefore, the restriction that primary conditions need to be partial variable assignments is not a true restriction of the expressivity of our formalism.

Proposition 2. *Let φ be any formula over the primary variables V_P. There exists a set of switched constraints C such that for every state s, $\langle p_\emptyset, C \rangle$ holds* *in s if and only if* $s[\varphi] = true$. *In addition, the size of C is polynomial in the size of* φ.

(Since there is no primary part of the partitioned condition is the empty partial variable assignment, figuring out whether $\langle p_\emptyset, C\rangle$ holds in s is equivalent to determining whether active(C, s) is satisfiable.)

Nebel [109] analyzed the complexity of compilations between classical planning formalisms spanning from (propositional) ADL to (propositional) STRIPS. Two of the implications of his results are that compiling away general Boolean formulas (in action preconditions and the goal) requires either a worst-case exponential increase in the size of the problem, or a super-linear (but still polynomial) increase in plan length. This partially answers the question of whether the increase in problem size when compiling away global state constraints (Proposition 1) is avoidable: If the constraint language is sufficiently expressive to compactly encode arbitrary action preconditions and goals over the primary variables – and linear switched constraints are, as shown by Proposition 2 – then compiling away those constraints must require at least the same complexity as compiling away those pre- and goal conditions.

3.4.1 Example

Here we will demonstrate the compilation of the invariant constraints and secondary conditions into a logical formula. We will use a small example from the power network domain (Section 3.3.2). Consider a small power network with only three power lines (and the corresponding switches y_1,y_2 and y_3), two buses with loads B_1 and B_2 (we denote their loads $\bar{l}_1$ and $\bar{l}_2$ and the "fed" statuses $f_{B,1}$ and $f_{B,2}$, respectively) and two generator buses G_1 and G_2 (with the generations g_1 and g_2). We denote the power in the lines p_1, p_2 and p_3. The susceptances of the lines are $\bar{b}_1 = \bar{b}_2 = \bar{b}_3 = 1$. The maximum generation in each if the buses is $\bar{g}_{1,max} = \bar{g}_{2,max} = 1$ and the loads of the loads of the non-generating buses are $\bar{l}_1 = \bar{l}_2 = 1$. The network is shown in Figure 3.2.

The state constraints are:

$$\begin{aligned}
y_1 = \mathit{true} &\rightarrow p_1 = -\bar{b}_1(\theta_{G,1} - \theta_{B,1})\\
y_1 = \mathit{true} &\rightarrow f_{G,1} = f_{B,1}\\
y_1 = \mathit{false} &\rightarrow p_1 = 0\\
y_2 = \mathit{true} &\rightarrow p_2 = -\bar{b}_2(\theta_{G,2} - \theta_{B,2})
\end{aligned}$$

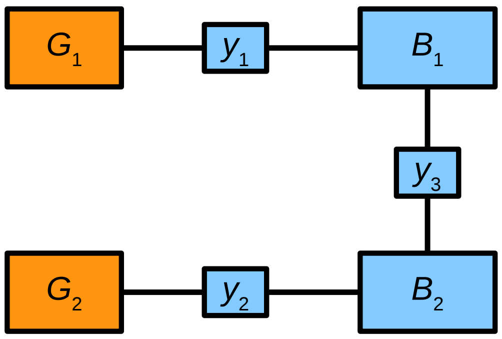


Figure 3.2: A small powernetowk.

$$\begin{aligned}
y_2 = \mathit{true} &\rightarrow f_{G,2} = f_{B,2} \\
y_2 = \mathit{false} &\rightarrow p_2 = 0 \\
y_3 = \mathit{true} &\rightarrow p_3 = -\bar{b}_3(\theta_{B,1} - \theta_{B,2}) \\
y_3 = \mathit{true} &\rightarrow f_{B,1} = f_{B,2} \\
y_3 = \mathit{false} &\rightarrow p_3 = 0
\end{aligned}$$

$$\begin{aligned}
&g_1 - p_1 = 0 \\
&f_{B,1} - p_1 + p_3 = 0 \\
&g_2 - p_2 = 0 \\
&f_{B,2} - p_2 - p_3 = 0 \\
&f_{G,1} = 1 \\
&f_{G,2} = 1 \\
&g_1 \leq 1 \\
&g_2 \leq 1
\end{aligned}$$

The goal is to supply both loads, so the goal constraints are $f_{B,1} = f_{B,2} = 1$.

As there are three switches there are three variables with binary domains and therefore 8 possible states, enumerated in the Figure 3.3. We can use the above given constraints to determine the validity of each of the states as well as whether the goal conditions are achieved in it. Using this, we can rewrite the goal as a formula $y_1 \wedge y_2$. The state validity can be expressed as $(y_1 \leftrightarrow y_2) \vee \neg y_3$ (the two invalid states violate the constraints as in each of them one generator feeds two loads).

y_1	y_2	y_3	Validity	Goal reached
F	F	F	T	F
F	F	T	T	F
F	T	F	T	F
F	T	T	F	F
T	F	F	T	F
T	F	T	F	F
T	T	F	T	T
T	T	T	T	T

Figure 3.3: Possible states.

3.5 Computing Heuristics

As seen in Section 2.2, relaxations of planning problems are used to derive admissible heuristics which are then used to guide search for optimal plans. Here we will adapt two well known relaxations used in classical planning to domains with numeric state constraints – monotone relaxation (see Section 2.2.1) and a form of abstraction called projection (see Section 2.2.3). The heuristics that we derive from the monotone relaxation are h_{max}, h^+ and an equivalent of LM-cut for problems with unit-cost actions. From abstraction we obtain the pattern-database heuristics. All of the relaxations below are developed for problems in which action costs are constant (meaning that the cost of an action is same regardless in which state is the action applied).

3.5.1 Monotone relaxations

The relaxation that we will describe in this section is an extension of MFDR (see Section 2.2.1) for planning with numeric state constraints. Primary variables function the same way as variables in MFDR, so defining the relaxed state is straightforward. As actions assign values only to primary variables, action effects function in the same way as in MFDR. Action preconditions and goals are partitioned conditions, so we will need to develop ways of determining whether they hold in a given relaxed state. We will define three different ways in which the partitioned conditions are evaluated and hence obtain three different relaxations. We will also discuss how these relaxations relate to each other.

Relaxed state

Here we will define the relaxed state which we will use for all three relaxations. This definition is analogous to Definition 6. As before a relaxed state associates a set of values (which is a subset of the variable's domain) with each primary variable.

Definition 20. *Let V_P be the set of primary state variables, and for each variable $v \in V_P$, with $\mathcal{D}(v)$ being the domain of v. A* relaxed state *s^+, is a mapping from V_P to sets of values such that $s^+[v] \subseteq \mathcal{D}(v)$ for all $v \in V_P$.*

Recall from Section 2.2.1 that a relaxed state s^+ represents a set of states, namely those obtainable by assigning each variable v_i one value from its value set $s^+[v_i]$:

$$\text{states}(s^+) = \{\{v_1 = x_1, \ldots, v_n = x_n\} \mid \forall i : x_i \in s^+[v_i]\}$$

In the same section, we have also seen that in the monotonic relaxation variables accumulate values as actions are applied, rather than switching between values as in the non-relaxed setting. Application of an action a in a relaxed state s^+ leads to a state $t^+ = s^+[\![a]\!]$ in which the sets of values associated with the variables are: $t^+[v] = s^+[v] \cup \{\mathit{eff}(a)[v]\}$ for all variables that appear in the effects of a and $t^+[v] = s^+[v]$ for all other variables. Of course, as in the non-relaxed case, a can only be applied if it is allowed in s^+ – we will explain how do we determine whether an action is allowed in a relaxed state later on.

As in classical planning, the value-set semantics extends to partial primary variable assignments (we will show that this has additional relevance in planning with numeric state constraints as triggers are partial primary variable assignments). Definition 7, which states that given a partial variable assignment φ, $s^+[\varphi]$ denotes the set of values that φ can take in relaxed state s^+. To paraphrase, *true* $\in s^+[\varphi]$ if and only if there exists a state $s \in \text{states}(s^+)$ such that $s[\varphi] = \mathit{true}$ (and analogously for *false*).

The relaxed planning problem

The relaxed planning problem Π^+ is defined by replacing the states with relaxed states (Definition 20), the applications of actions with the relaxed application of actions and evaluation of partitioned conditions (that is, action preconditions and goals) with one of the three possible ways of evaluating whether a partitioned condition holds in a relaxed state. Depending on which of those three definitions are used, we obtain either *weak relaxation*, *intermediate relaxation* or *strong relaxation*. We denote the resulting relaxed planning problems $\Pi^+{}_{weak}$, $\Pi^+{}_{intermediate}$ or $\Pi^+{}_{strong}$, respectively. The three ways partitioned conditions are evaluated will be given in the following sections.

As with MFDR, any plan for the original problem is also a plan under relaxed semantics and, therefore, the minimal relaxed plan cost is also a lower bound on minimal real plan cost (this is true regardless which of the three relaxations are used). The key property ensuring this is that the relaxation is monotonic – for any partial variable assignment φ and a relaxed state s^+, if $e \in s^+[\varphi]$ with e being either true or false, then $e \in t^+[\varphi]$ for any relaxed state t^+ reachable from s^+ by relaxed action application. Also, any partitioned condition that holds in s^+ still holds in t^+ (in a delete relaxation we say that "true conditions remain true").

The monotonicity property of the relaxation means that we can build a relaxed planning graph, following the same procedure as in MFDR (see explanation in Section 2.2.2). The relaxed planning graph consists of alternating relaxed states (they are analogous to the fact layers in delete relaxation) and action layers. Each action layer includes all actions that are allowed in the preceding relaxed state and that have not appeared in any previous action layer. An action a is allowed in a relaxed state s^+ iff (i) $\langle \mathrm{pre}(a)_P, \mathrm{pre}(a)_S \rangle$ holds in s^+ and (ii) $\langle \mathrm{eff}(a), \emptyset \rangle$ holds in $s^+[\![a]\!]$. (We will formulate three different ways of determining whether a partitioned condition holds in a relaxed state, leading to three relaxations of different strength. These will be presented in the following sections.) The second part ensures that the action's effects, considered by themselves, do not lead to an invalid state. Note that just as in the classical relaxed planning graph, we make an independence assumption in that the allowableness of each action is tested separately from other actions in the same layer. The next fact layer is the relaxed state that results from applying the effects of all actions in the current layer. (There is no need for explicit no-ops, since previously achieved values remain under the value accumulating semantics.) Graph construction stops when the goal holds in the last relaxed state, or when two consecutive relaxed states are the same, indicating that all reachable variable values have been achieved. This process provides a procedure for determining the relaxed plan existence – if it ends with a final relaxed state in which the goal does not hold, the goal is not relaxed reachable.

The size of the relaxed planning graph is polynomial with respect to the size of the planning problem. There are two ways to show that this is the case. First, we are never required to apply any action more than once, so the number of layers representing the relaxed states is at most the number of actions in the relaxed planning problem Π^+ plus one (representing the initial relaxed state). This also bounds the length of optimal plans for Π^+ by the number of actions in the problem. (However, computing the optimal cost of a plan for the relaxed problem is NP-hard [20].)

Another way to demonstrate this property is to note that since the values accumulate with relaxed action application, in the final relaxed state every variable can end up having at most the number of values as there are in its domain, so the set of values associated with the variable can grow at most the number of times equal to the size of its domain minus one (due to the initial value in the set). This means that the number of layers is bounded by $1 + \sum_{v \in V}(|\mathcal{D}(v)| - 1)$.

The weak monotone relaxation

We start with the weakest relaxation. The key question is how to evaluate conditions on secondary variables in a relaxed state. We treat this as a question of consistency, which can be delegated to an appropriate external solver. As mentioned in Section 3.2, the syntactic form of switched constraints provides a natural interface for this separation of concerns, wherein the constraint solver only needs to consider the secondary part of the active subset of constraints. We therefore need to figure out which of the constraints are active, given a relaxed state. The following definition is used in the weak and the intermediate relaxations.

Definition 21. *For a relaxed state s^+ and a set of switched constraints C, the relaxed active constraints of C in s^+ are*

$$\text{active}^+(C, s^+) = \{\gamma \mid \varphi \to \gamma \in C, \mathit{false} \notin s^+[\varphi]\}$$

That is, a switched constraint $\varphi \to \gamma$ is active in a relaxed state only if the triggering condition φ cannot evaluate to *false* in s^+ – i.e. φ *must* be true.

As in the non-relaxed case, in order for a partitioned condition to hold in a given state, both primary and secondary conditions need to hold in that state. The primary condition holds if there is at least one state in a set of states represented by the relaxed state in which the primary condition holds. For the secondary condition, we use the Definition 21 to figure out which of the constraints are active and then we determine whether the active constraints are satisfiable.

Definition 22. *The* weak monotone relaxation *of a planning problem with global state constraints replaces the evaluation of partitioned conditions with the following condition: a partitioned condition $\langle \varphi_P, \varphi_S \rangle$ holds in s^+ iff (i) $\mathit{true} \in s^+[\varphi_P]$ and (ii) $\text{active}^+(\varphi_S \cup C_{\mathsf{inv}}, s^+)$ is satisfiable.*

The relaxation obtained using this definition is monotonic. As more actions are applied, sets of values associated with each of the variables grows. I.e. applying an action can only add a new value to a set associated with a variable and cannot remove a value. Consequently, the set of active constraints can only shrink as more actions are applied. Applying an action cannot activate a constraint that was previously inactive. Monotonicity applies to partitioned conditions as well:

Proposition 3. *Let s^+ be a relaxed state and t^+ be a state that results from the application of a sequence of actions $a_1, ..., a_n$ in s^+. Let $\langle c_P, c_S \rangle$ be a partitioned condition. If $\langle c_P, c_S \rangle$ holds in s^+, it also holds in t^+.*

Proof. Monotonicity of the primary condition c_P follows directly from the monotonicity of the classical relaxation. Let $\varphi \rightarrow \gamma$ be a switched constraint in c_S. Since the set of values grows monotonically, $s^+[\varphi] \subseteq t^+[\varphi]$. Therefore, if *false* $\in s^+[\varphi]$ then *false* $\in t^+[\varphi]$. As $\varphi \rightarrow \gamma$ is inactive in any relaxed state in which *false* is in the set of values associated with φ, the set of active constraints in t^+ is a subset of the set of active constraints in s^+, $\text{active}^+(c_S, t^+) \subseteq \text{active}^+(c_S, s^+)$. As removing a constraint from a set cannot cause a contradiction, if $\text{active}^+(c_S, s^+)$ is satisfiable, then $\text{active}^+(c_S, t^+)$ is also satisfiable. □

This relaxation also has the property that invariant constraints are always satisfiable in any relaxed state that can be reached starting from a valid state (this is because applying an action can only remove constraints from the set of active constraints as mentioned earlier). Thus, in this relaxation, if an action's precondition holds in a relaxed state, the action is also allowed in that relaxed state. This eliminates one of the two consistency tests needed for each action in each layer of the relaxed planning graph. If the action has no secondary preconditions, no consistency test is needed; it suffices to check if the primary precondition holds in the relaxed state.

If the consistency of a set of constraints can be determined in polynomial time (as is the case when the constraints are linear), the allowedness of an action can be determined in polynomial time as well. Therefore, the relaxed planning graph (either for reachability testing or computing h_{max}), can be constructed in polynomial time.

In Section 2.3.2, we described how Francès and Geffner [51] use state constraints as a way to strengthen the monotonic relaxation. Here we will demonstrate that same can be done in our framework. Consider, for example, an instance of the Counters domain with three (integer) counters, X_1, X_2 and X_3, initially all at zero, and the goal $\{X_1 < X_2, X_2 < X_3\}$ (domain described in Section 3.3.4). In our formulation of the problem, relaxed application of the actions $\mathsf{inc}(2)$ and $\mathsf{inc}(3)$ leads to a relaxed state s^+ where: $s^+[p_{1,0}] = s^+[p_{2,0}] = s^+[p_{3,0}] = \{true\}$, $s^+[p_{1,1}] = s^+[p_{2,2}] = s^+[p_{3,2}] = \{false\}$, and $s^+[p_{2,1}] = s^+[p_{3,1}] = \{true, false\}$. Hence, we have the active constraints $0 \leq x_1 \leq 0$, $0 \leq x_2 \leq 1$ and $0 \leq x_3 \leq 1$. (Recall that the secondary variables x_i here are real-valued, while the counters X_i are integer-valued.) We can see that their conjunction with the secondary goal condition $\{x_1 + 1 \leq x_2, x_2 + 1 \leq x_3\}$ is not satisfiable. Therefore, although both of the individual goal conjuncts are relaxed achievable by the plan $\mathsf{inc}(2)$, $\mathsf{inc}(3)$, their conjunction is not.

The intermediate monotone relaxation

The intermediate monotone relaxation is more computationally expensive, but it can give higher bounds on the heuristic cost estimate.

Consider an example from Hydraulic Blocks World domain described in Section 3.3.1. In the state depicted in Figure 3.1(a), the variable $\mathsf{in}_\mathsf{B} = 2$. Applying the action $\mathsf{pickup}_{\mathsf{B},2}$ to this relaxed state results in a new relaxed state with $s^+[\mathsf{in}_\mathsf{B}] = \{2, \mathsf{none}\}$. We need to take into account the invariant constraints (HBW.b.i–HBW.b.ii):

$$\mathsf{in}_i \neq k \rightarrow p_{i,k} = 0 \qquad \text{(HBW.b.i)}$$
$$\mathsf{in}_i = k \rightarrow p_{i,k} = \bar{w}_i \qquad \text{(HBW.b.ii)}$$

with $k = 2$, $i = \mathsf{B}$ and $\bar{w}_\mathsf{B} = 2$. We wish to test whether the resulting set of constraints is consistent with the query $p_\emptyset \rightarrow p_{\mathsf{B},2} = 1$.

We obtain the following set of constraints:

$$\mathsf{in}_\mathsf{B} \in \{2, \mathsf{none}\} \qquad \text{(Example.1.i)}$$
$$\mathsf{in}_\mathsf{B} \neq 2 \rightarrow p_{\mathsf{B},2} = 0 \qquad \text{(Example.1.ii)}$$
$$\mathsf{in}_\mathsf{B} = 2 \rightarrow p_{\mathsf{B},2} = 2 \qquad \text{(Example.1.iii)}$$
$$p_\emptyset \rightarrow 0 \leq p_{\mathsf{B},2} \leq 2 \qquad \text{(Example.1.iv)}$$
$$p_\emptyset \rightarrow p_{\mathsf{B},2} = 1 \qquad \text{(Example.1.v)}$$

In the relaxed state above the variable in_B is associated with the set of values $\{2, \mathsf{none}\}$, meaning that the relaxed state represents the set of all states in which the value of in_B is in that set. In each of those states, in_B has only one value – either 2 or none. Assigning either of those values to in_B results in a non-satisfiable set of constraints, meaning that those states are not consistent with the query $p_\emptyset \rightarrow p_{\mathsf{B},2} = 1$. However, given Definition 22 (weak relaxation), the constraints $\mathsf{in}_\mathsf{B} \neq 2 \rightarrow p_{\mathsf{B},2} = 0$ and $\mathsf{in}_\mathsf{B} = 2 \rightarrow p_{\mathsf{B},2} = 2$ are both inactive and the set of constraints is satisfiable. Hence, although the query $p_\emptyset \to\ p_{\mathsf{B},2} = 1$ is not satisfiable in any of the states in s^+, it is consistent if we use the Definition 22 to determine the active subset. (This also demonstrates why the constraint (HBW.b.iv), which is redundant in the non-relaxed problem, is useful in the relaxation: without it, $p_{\mathsf{B},2}$ would be free to take any value in s^+.)

As a way to partially rectify this, we present the *intermediate monotone relaxation.* This relaxation uses the concept of conditioned relaxed state.

Definition 23. *Let s^+ be a relaxed state and φ a partial primary variable assignment such that true $\in s^+[\varphi]$. $s^+|\varphi$, called s^+* conditioned *on φ, is the relaxed state defined by*

$$\begin{aligned} (s^+|\varphi)[v] &= \{e\} && \textit{if } \varphi[v] = e \\ (s^+|\varphi)[v] &= s^+[v] && \textit{otherwise} \end{aligned}$$

It is obvious that states$(s^+|\varphi) \subseteq$ states(s^+) for any φ because $s^+|\varphi[v] \subseteq s^+[v]$ for all v.

Definition 24. *We define the* intermediate monotone relaxation *of a planning problem with global state constraints by replacing the evaluation of partitioned conditions with the following – a partitioned condition $\langle \varphi_P, \varphi_S \rangle$ holds in s^+ iff (i) true $\in s^+[\varphi_P]$ and (ii)* active$^+(\varphi_S \cup C_{\mathsf{inv}}, s^+|\varphi_P)$ *is satisfiable.*

Conditioning the relaxed state on the primary part of an action's precondition (or its effects when testing if the resulting relaxed state is valid) asserts the variable–value equalities that are known to be necessarily true while evaluating the triggering conditions of switched constraints and therefore strengthens the relaxation. For example, to determine if the action $\mathsf{stack}_{\mathsf{B},\mathsf{A},1}$ is allowed in the relaxed state s^+ in the example above, we test the satisfiability of the invariant constraints that are active in the resulting state condition on this action's effects. The action has the effect $\mathsf{in}_{\mathsf{B}} = 1$, so in the resulting state $t^+ = s^+[\![\mathsf{stack}_{\mathsf{B},\mathsf{A},1}]\!]$, $t^+[\mathsf{in}_{\mathsf{B}}] = \{1, 2, \mathsf{none}\}$. However, $(t^+|\text{eff}(\mathsf{stack}_{\mathsf{B},\mathsf{A},1}))[\mathsf{in}_{\mathsf{B}}] = 1$, activating constraints $\mathsf{in}_{\mathsf{B}} = 1 \to p_{\mathsf{B},1} = 2$, $\mathsf{in}_{\mathsf{B}} \neq 2 \to p_{\mathsf{B},2} = 0$, etc., such that the resulting set of active secondary constraints is unsatisfiable, proving that the sequence of actions $\mathsf{pickup}_{\mathsf{B},2}$, $\mathsf{stack}_{\mathsf{B},\mathsf{A},1}$ is not valid in the intermediate relaxation of the problem.

It should be noted that the difference between the weak and the intermediate monotone relaxation exists only in domains in which some of the reachable states are invalid (such as the Hydraulic Blocks World and the Switching Problems in Power Networks). In the two other example domains, namely the Long-haul and Counters domains, the secondary model determines only whether the goal has been achieved (and there is no primary goal condition) and no reachable state is invalid. In those domains the intermediate and the weak relaxation are equally strong.

As with the weak relaxation, whether a partitioned condition holds can be decided in polynomial time (provided that the consistency of the set of constraints can be determined in polynomial time).

The strong monotone relaxation

Finally, we will show how the strong relaxation works. Here a set of constraints holds in a relaxed state s^+ only if it is consistent with the restrictions on the primary variable assignment imposed by s^+. To paraphrase, in a strong relaxation a partitioned condition holds if and only if the set states(s^+) contains at least one state in which the partitioned condition holds. Returning to the above HBW example (Figure 3.1), we note that the primary variable in_B can either take the value of 2 or none. The set of constraints Example.1.i–Example.1.v is not satisfiable, since s^+ encodes a discrete disjunction between $\mathsf{in}_\mathsf{B} = 2 \wedge p_{\mathsf{B},2} = 2$ and $\mathsf{in}_\mathsf{B} = \mathsf{none} \wedge p_{\mathsf{B},2} = 0$. That is, no state consistent with these constraints exists within states(s^+).

Definition 25. *The* strong monotone relaxation *of a planning problem with global state constraints replaces the evaluation of partitioned conditions with the following condition: a partitioned condition $\langle \varphi_P, \varphi_S \rangle$ holds in s^+ iff $\{\varphi_P\} \cup \{v \in s^+[v] \mid v \in V_P\} \cup \varphi_S \cup C_{\mathsf{inv}}$ is satisfiable.*

Note that if $\{\varphi_P\} \cup \{v \in s^+[v] \mid v \in V_P\}$ is satisfiable, then *true* $\in s^+[\varphi_P]$ by definition; hence, we omit this condition. We also make the following observation regarding the relationship between strong and intermediate relaxations.

Proposition 4. *Let V_P and V_S be sets of primary and secondary state variables, respectively. Let $\langle \varphi_P, \varphi_S \rangle$ be a partitioned condition, and s^+ a relaxed state. Then $\langle \varphi_P, \varphi_S \rangle$ holds in s^+ under strong relaxation only if there is an assignment to V_S satisfying* active$^+(\varphi_S \cup C_{\mathsf{inv}}, s^+|\varphi_P)$.

Proof. Suppose $\langle \varphi_P, \varphi_S \rangle$ holds in s^+, i.e., that $\{\varphi_P\} \cup \{v \in s^+[v] \mid v \in V_P\} \cup \varphi_S \cup C_{\mathsf{inv}}$ is satisfiable (Definition 25); let σ be a satisfying assignment over $V_P \cup V_S$. If σ restricted to V_S does not satisfy active$^+(\varphi_S \cup C_{\mathsf{inv}}, s^+|\varphi_P)$ there must be a switched constraint $\psi \to \gamma \in$ active$^+(\varphi_S \cup C_{\mathsf{inv}}, s^+|\varphi_P)$ such that γ is false under assignment σ. Let s' be σ restricted to the primary variables, V_P. Because σ satisfies $\varphi_S \cup C_{\mathsf{inv}}$, it must then be the case that the triggering condition ψ is false under σ, i.e., $s'[\psi] =$ *false*. Because σ satisfies $\{\varphi_P\} \cup \{v \in s^+[v] \mid v \in V_P\}$, s' is a state in states$(s^+|\varphi_P)$. Hence, *false* $\in s^+|\varphi_P[\psi]$, contradicting that $\psi \to \gamma \in$ active$^+(\varphi_S \cup C_{\mathsf{inv}}, s^+|\varphi_P)$. □

Comparing the relaxations

Proposition 5. *Given a planning problem* Π*, any plan for its strong relaxation* $\Pi^{+}{}_{strong}$ *is also a plan for its intermediate relaxation* $\Pi^{+}{}_{intermediate}$.

Proof. Any partitioned condition $\langle \varphi_P, \varphi_S \rangle$ that holds in a given relaxed state s^+ under the strong relaxation also holds in the same relaxed state under the intermediate relaxation. Referring to Definition 24, this is because (i) if φ_P holds in s^+ under strong relaxation, *true* $\in s^+[\varphi_P]$ is true by definition (see above) and (ii) if $\langle \varphi_P, \varphi_S \rangle$ holds in s^+ under the strong relaxation, then it also holds in the intermediate one due to the Proposition 4. Therefore any condition, such as goal or action precondition, that holds in s^+ under the strong relaxation also holds in the intermediate relaxation and any plan for the strong relaxation of a planning problem is also a plan for the intermediate monotone relaxation. □

Proposition 6. *Given a planning problem* Π*, any plan for its intermediate relaxation* $\Pi^{+}{}_{intermediate}$ *is also a plan for its weak relaxation* $\Pi^{+}{}_{weak}$.

Proof. If $\langle \varphi_P, \varphi_S \rangle$ holds in s^+ under intermediate relaxation, it also holds s^+ under the weak relaxation. Comparing Definition 24 and Definition 22, we see that the requirement (i), *true* $\in s^+[\varphi_P]$, is identical under both relaxations. As for the requirement (ii), given that $\text{states}(s^+|\varphi) \subseteq \text{states}(s^+)$ (see above), it follows that $\text{active}^+(\varphi_S \cup C_{\mathsf{inv}}, s^+) \subseteq \text{active}^+(\varphi_S \cup C_{\mathsf{inv}}, s^+|\varphi_P)$. Given that inactivating a constraint cannot make a previously consistent set inconsistent, if $\text{active}^+(\varphi_S \cup C_{\mathsf{inv}}, s^+|\varphi_P)$ is satisfiable, then $\text{active}^+(\varphi_S \cup C_{\mathsf{inv}}, s^+)$ has to be satisfiable as well. Any condition, such as action preconditions and goal, that holds in s^+ under the intermediate relaxation also holds in under s^+ under the weak relaxation. Therefore, any plan for the intermediate relaxation is also a plan for the weak relaxation. □

A consequence of these two propositions is that the optimal plan cost for the problem under the intermediate relaxation is a lower bound on the cost of the strong relaxation and the optimal plan cost under the weak relaxation is a lower bound for the cost of the intermediate relaxation.

Computational costs

An important difference between the three above mentioned relaxations are their computational costs. As already stated, the size of the relaxed planning graph is polynomial regardless of the relaxation used.

Despite this, its construction is not necessarily tractable if we use the strong relaxation. As shown in Section 3.4, the secondary model can encode constraints that are equivalent to non-simple conditions over the primary state variables. Thus, evaluating a partitioned condition in a relaxed state according to Definition 25 is NP-hard. This is true even if all state constraints are linear switched constraints, because Definition 25 asks for an assignment to the discrete primary variables that are under-constrained by the relaxed state. We prove this by reducing a zero-one integer program (with no optimisation) to the problem of finding an assignment to a relaxed state s^+ (note that finding an assignment consistent with the constraints is equivalent to finding a single state in states(s^+)).

Proposition 7. *Finding an assignment of variables consistent with a relaxed state s^+ is NP-hard.*

Proof. The problem of finding an assignment of $x \in \{0,1\}^n$ that is consistent with the constraints $Ax \leq b$ where A is a $m \times n$ matrix and b is an m-vector is NP-hard [90]. We introduce a primary variable p_i for each $x_i \in x$ and set of switched constraints that restricts each variable to $\{0,1\}$ domain. We define a relaxed state s^+ where $s^+[p_i] = \{\mathit{true}, \mathit{false}\}$ for all p_i. This allows us to express this problem as finding an assignment consistent with:

$$\begin{array}{rll} & Ax \leq & b \\ p_i = \mathit{true} \rightarrow & x_i = & 1 \\ p_i = \mathit{false} \rightarrow & x_i = & 0 \\ & s^+[p_i] = & \{\mathit{true}, \mathit{false}\} \\ & \text{for} & i = 1, \ldots, n \end{array}$$

Therefore, by finding the consistent assignment to the relaxed state, we can solve the original problem. It follows that determining the consistency of a relaxed state under the strong relaxation is NP-hard. □

Using the intermediate relaxation makes the relaxed planning graph construction tractable. This is because determining the satisfiability is decidable in polynomial time, assuming that the consistency of the secondary constraints is tractable (as is the case with systems of linear equations).

However, although tractable, the intermediate monotone relaxation can still be too computationally expensive to be the basis of a cost-effective search heuristic. It needs two secondary constraint consistency checks for each action that may be added to each layer of the relaxed planning graph. Each consistency check involves a call to an external solver, which is substantially more time-consuming than evaluating a partial variable assignment over the primary variables in a relaxed state. In contrast, the weak relaxation only requires one secondary constraint consistency check for each action whose applicability we are testing (or zero, if an action has no secondary preconditions).

3.5.2 Abstraction-based heuristics

In Section 2.2.3 we discussed PDB, which is an abstraction-based heuristic. Here, we will explain how can PDBs be adapted for planning with state constraints.

Recall that we can view an abstract state s^A as a relaxed state in which variables in V^A have only a single value and variables not in A have all possible values in their domain. Thus, the truth value of a partitioned condition in an abstract state is determined in the same way as in any other relaxed state, according to Definition 22, or one of the two stronger relaxations.

In the classical setting, a PDB can be efficiently constructed by an exhaustive reverse exploration from the (abstract) goal states, but this strategy is not easily adapted to problems with global state constraints. For example, in most of the example domains presented in Section 3.3, the goal condition is defined on the secondary model only. This means to even identify the abstract goal states we need to enumerate the models of the set of constraints over primary and secondary variables.

Instead, we create PDBs using the following two step method:

1. In the first stage, we build an explicit graph of the reachable abstract space by forward exploration. In this stage we also identify which of the abstract states satisfy the goal.

2. Next, we compute the optimal costs to reach the goal for each of the abstract states generated in the previous step. This is done by reverse exploration, same as in the classical case.

This is more expensive than the classical PDB construction, but still practical for projections that induce a small enough abstract state space. Once the PDB is built, however, state evaluation is done by a table lookup, and takes no more time than in a standard PDB heuristic.

3.5.3 Computation of h_{max}

The relaxed planning graph (introduced in Section 2.2.2 can be constructed under all three monotone relaxations, and provides a basis for computing several admissible heuristics. As already discussed, it provides a relaxed reachability test, which enables us to compute h^+ using the iterative landmark algorithm (described below). The number of action layers needed before the goal condition holds is a lower bound on optimal plan length. If all actions have unit cost, it is also a lower bound on plan cost, and analogous to the

$h_{\max}$ heuristic [11, 16]. A cost-sensitive version of this heuristic can be obtained by indexing fact layers of the relaxed planning graph by cost, rather than depth.

This is done in the following way (see Figure 3.4) – given a relaxed planning graph, the cost of a relaxed state $\mathit{cost}(s^+)$ is defined as the sum of the costs of layers of actions that lead to that state, with the initial state having the cost of zero, $\mathit{cost}({s^+}_0) = 0$. For a relaxed state s^+, there is a subset of actions that are applicable in s^+, which we denote A_{s^+}. This set of actions is divided into subsets by cost, $A_{s^+,c_1}, \ldots, A_{s^+,c_n}$, with costs $c_{s^+,1}, \ldots, c_{s^+,n}$.

Given ${s^+}_0$, the construction of the relaxed planning graph starts by applying all the actions its cheapest set, $A_{{s^+}_0,c_1}$, creating a new relaxed state ${t^+}_1$, whose cost is $\mathit{cost}({t^+}_1) = \mathit{cost}({s^+}_0) + c_{{s^+}_0,1}$. The values of variables in ${t^+}_1$ are determined using the rules for relaxed action application, ${t^+}_1[v] = {s^+}_0[v] \cup \{\mathit{eff}(a)[v]\}$ for variables v that appear in the effects of the actions $a \in A_{s^+,c_1}$, and ${t^+}_1[v] = {s^+}_0[v]$ for all the other variables (if some variable appears in more than one action, all of the values are added to the set). Whenever an action is applied, it can be discarded, as we never have to apply any action more than once.

We build the relaxed planning graph by always adding the next cheapest relaxed state. Whenever we generate a relaxed state, we identify the applicable set of actions, e.g. $A_{{t^+}_1}$ for ${t^+}_1$, and divide it into subsets by cost, $A_{t^+,c_1}, \ldots, A_{t^+,c_m}$. We then compare the costs of the all the relaxed states that could potentially be created, considering all the states currently in the graph and their corresponding sets of (still unused) applicable actions, and generate the one with the minimum cost. E.g., $\mathit{cost}({t^+}_1) + c_{t^+,1} \leq \mathit{cost}({s^+}_0) + c_{s^+,2}$, then the state u_1 is created by applying actions in A_{t^+,c_1} to ${t^+}_1$. Otherwise, the next cheapest state, ${t^+}_2$, is created by applying actions in $A_{{s^+}_0,c_2}$ to ${s^+}_0$. For each variable v, the set of values appearing in any relaxed state is the union of the values in the effects of all actions used to generate the relaxed state and the most expensive state of lower (or equal) cost. E.g., if ${u^+}_1$ is cheaper than (or of equal cost as) ${t^+}_2$, the effects of the actions in A_{t^+,c_1} will appear in ${t^+}_2$ (together with the values appearing in the effects of actions in $A_{{s^+}_0,c_2}$). If two relaxed states end up having equal costs, we simply combine them by assigning each variable a union of values held in those two relaxed states.

We keep adding relaxed states in this manner until we run out of actions, in which case the goal is not relaxed reachable (and the non-relaxed problem is unsolvable), or we find a relaxed state in which the goal holds (whichever comes first). The value $h_{\max}$ is the cost of the cheapest relaxed state which achieves the goal. If the goal is not relaxed reachable, $h_{\max}$ is infinite.

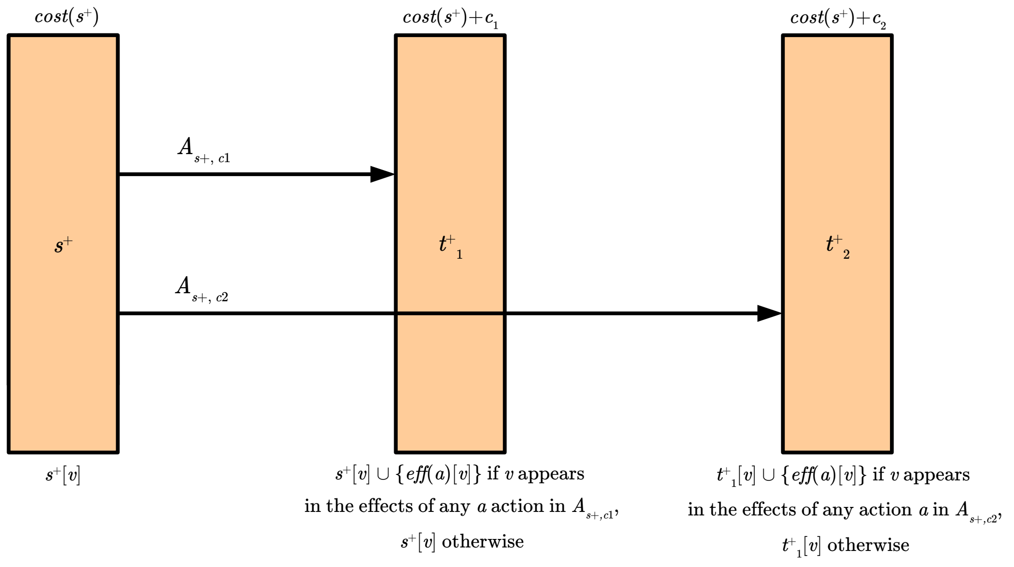

$cost(s^+)$
$cost(s^+)+c_1$
$cost(s^+)+c_2$
$A_{s+,\ c1}$
$A_{s+,\ c2}$
s^+
$t^+{}_1$
$t^+{}_2$
$s^+[v]$
$s^+[v] \cup \{eff(a)[v]\}$ if v appears in the effects of any a action in $A_{s+,c1}$, $s^+[v]$ otherwise
$t^+{}_1[v] \cup \{eff(a)[v]\}$ if v appears in the effects of any action a in $A_{s+,c2}$, $t^+{}_1[v]$ otherwise

Figure 3.4: Building a relaxed planning graph. The relaxed state $t^+{}_1$ is obtained by applying the actions in set A_{s^+,c_1} and the relaxed state $t^+{}_2$ is obtained by applying the actions in the set A_{s^+,c_2}. The cost of $t^+{}_1$ is $cost(s^+)+c_1$, where c_1 is the cost of each of the actions in A_{s^+,c_1}. The cost of $t^+{}_2$ is $cost(s^+)+c_2$, where c_2 is the cost of each of the actions in A_{s^+,c_2}. Since $c_1 < c_2$, for each variable v, the set of values associated with v in $t^+{}_2$ contains all of the values in $t^+{}_1[v]$ as well as the values added due to the effects of the actions in A_{s^+,c_2}.

3.5.4 Computation of h^+

In Section 2.2.2 we introduced the *optimal delete relaxation heuristic*, or h^+, whose value is the cost of the optimal relaxed plan (in our case under monotone, rather than delete relaxation). Although it is NP-hard to compute, it dominates all the heuristics based on the delete relaxation [14]. To compute h^+, we use the algorithm by Haslum, Slaney and Thiébaux [74], which we will briefly describe below. The algorithm finds a set of *disjunctive action landmarks* [91] such that a *minimum-cost hitting set* over this collection is an optimal relaxed plan and computes its cost. Because this algorithm interfaces with the planning formalism only through the relaxed reachability test[3], which we can perform as explained above, and because our relaxation, like the classical delete-relaxation, does not require any action more than once in an optimal relaxed plan, this algorithm can be applied unchanged to compute h^+ in our setting as well.

Disjunctive landmark algorithm

A *disjunctive action landmark* (or a landmark, for short) of a problem Π^+ is a set of actions such that at least one action in the set has to be included in any valid plan for Π^+. Consequently, for any collection of landmarks L for Π^+, any valid plan for Π^+ contains at least one action from each landmark in $\mathcal{L}$.

The simple algorithm for finding the collection of landmarks $\mathcal{L}$ and computing h^+ is shown in Figure 3.5. Initially, the collection of landmarks $\mathcal{L}$ and a set of action A are empty.

NewLandmark takes a set of actions A as an input and returns an inclusion-minimal landmark. By *inclusion minimal landmark* we mean that no proper subset of the returned landmark is also a landmark. We know that given any set of actions A such that the goal is *not* relaxed-reachable with actions in A, the complement $\overline{A}$ of A (with respect to the whole set of actions of the problem Π^+), is a disjunctive action landmark for Π^+ (as $\overline{A}$ has to contain at least one action that is necessary to make the goal reachable). If A is *inclusion-maximal*[4], then $\overline{A}$ is inclusion-minimal landmark (since if some proper subset of $\overline{A}$ were also a landmark, A would not be maximal).

Therefore, we take an action a from $\overline{A}$ and test whether the goal is reachable with $A \cup \{a\}$ (reachability testing is performed by building a relaxed

[3]That is, to determine whether the goal is relaxed-reachable from the initial state using a given subset of actions.

[4]By inclusion-maximal, or maximal with the respect to set inclusion, we mean that no proper superset of A has the same properties – i.e. the goal is relaxed reachable with every proper superset of A.

```
1: procedure ITERATIVE LANDMARK ALGORITHM
2:   $\mathcal{L} = \emptyset$
3:   $A = \emptyset$
4:   while Goal not reachable with actions in $A$ do
5:     $\mathcal{L} = \mathcal{L} \cup$ NEWLANDMARK($A$)
6:     $A =$ MINCOSTHITTINGSET($\mathcal{L}$)
     $h^+ = \sum_{a \in A} cost(a)$
7:   return $h^+$
```

Figure 3.5: Iterative Landmark Algorithm.

planning graph). Iff adding a *does not* make the goal reachable, A is assigned $A \cup \{a\}$ and a is removed from $\overline{A}$. After testing all of the actions that were initially in $\overline{A}$, the actions remaining in $\overline{A}$ form a landmark, as adding any of those actions to A would make the goal reachable using A.

MINCOSTHITTINGSET is a procedure for finding the minimum-cost hitting set over a collection of landmarks. The definition of a hitting set which we give here is adapted from Bonet and Helmert [14]. Let $A = \{a_1, \dots, a_n\}$ be a set and $\mathcal{F} = \{F_1, \dots, F_m\}$ a collection of subsets of A. A subset $H \subseteq A$ is a hitting set iff $H \cap F_i \neq \emptyset$ for all $1 \leq i \leq m$ (i.e. H hits each set F_i). If each $a \in A$ is associated with a cost, then the cost of a hitting set is $\sum_{a \in H} cost(a)$, where $cost(a)$ is a constant cost of an action a (recall that our heuristic only works for problems in which the cost of an action does not depend on the state the action is applied in). In our implementation, we formulate the problem of finding a cost-optimal hitting set as an integer programming problem and we use Gurobi to solve it.

The algorithm terminates when the goal is reachable using only actions in A. (Of course, if the goal cannot be reached with all of the actions of Π^+, the problem is unsolvable and hence h^+ is infinite.) As any plan for the relaxed problem must contain an action from every landmark in $\mathcal{L}$, the minimum cost hitting set has to be the cheapest possible relaxed plan and therefore the sum of the costs of actions in A is the cost of the optimal relaxed plan h^+.

We can use information from the parent state to speed up the iterative landmark algorithm computation. Let s be a state, $\mathcal{L}(s)$ the collection of landmarks found for s, and s' the state resulting from applying action a in s. Then each element of $\{L \in \mathcal{L}(s) \mid a \notin L\}$ is also a landmark for s'. Thus, we can start the algorithm with this collection of landmarks, instead of an empty collection. This reduces the number of iterations, and hence the number of relaxed reachability tests substantially. A similar technique was used by Pommerening and Helmert [117] for the LM-Cut heuristic.

```
1: procedure DISJOINT LANDMARK ALGORITHM
2:   $L = \emptyset$
3:   $A = \emptyset$
4:   while Goal not reachable with actions in $A$ do
5:     $\mathcal{L} = \mathcal{L} \cup$ NEWLANDMARK($A$)
6:     $A = \bigcup_{L \in \mathcal{L}} l$
7:   $h_{DLA} = \sum_{L \in \mathcal{L}} \min_{a \in L} cost(a)$
8:   return $h_{DLA}$
```

Figure 3.6: Disjoint Landmark Algorithm.

The algorithm given in Figure 3.5 can be improved in several other ways, which are explained in the paper by Haslum et al. [74].

3.5.5 A weaker approximation of h^+

As noted in Section 2.2.2, given that h^+ is computationally expensive, we have explored computing admissible approximations. By restricting Algorithm 3.5 to generating only *disjoint landmarks*, we obtain a faster-to-compute but potentially weaker heuristic. If all the actions have binary costs, this heuristic becomes equivalent to the LM-Cut.

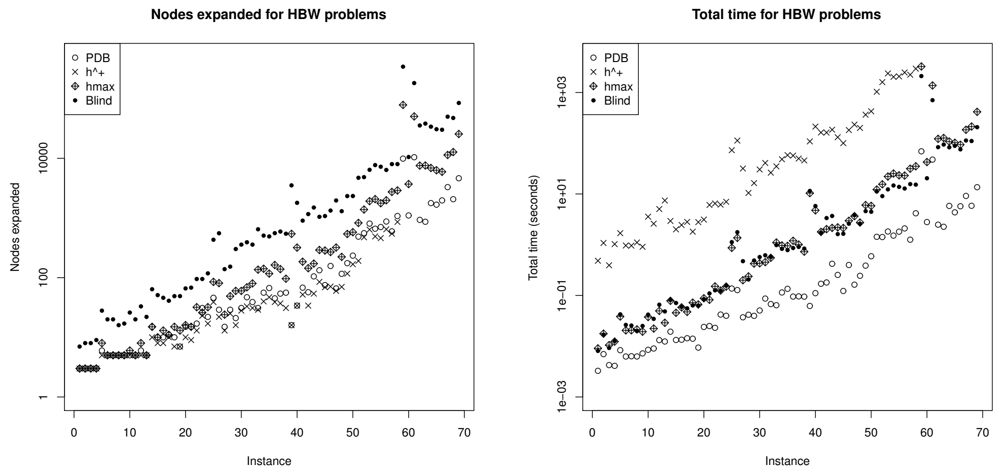
Our new algorithm (Figure 3.6) differs from the algorithm we used to compute h^+ in that instead of calculating the minimum-cost hitting set, we use the union of all of the landmarks generated so far (Line 6) as an argument for the NEWLANDMARK procedure. We stop when the goal is reachable with the set $A = \bigcup_{L \in \mathcal{L}} L$. Since we are using the union of all the landmarks as a starting point for generating a new landmark, landmarks obtained this way are *disjoint* – that is, for all $i \neq j$, $L_i \cap L_j = \emptyset$.

As each of the landmarks contains at least one of the actions that have to be included in every solution of a delete-relaxed problem, the sum of the minimum action costs in all landmarks places a lower bound on h^+. This value is therefore our heuristic cost estimate:

$$h_{DLA} = \sum_{L \in \mathcal{L}} \min_{a \in L} cost(a)$$

For unit cost actions, the heuristic cost estimate simply becomes the number of landmarks $\mid \mathcal{L} \mid$ generated. As with h^+, we can reuse the collection of landmarks generated in a given state when we are computing the h_{DLA} of its children.

Recall from Section 2.2.2 that if the planning problem is limited to binary cost actions, the landmarks generated by LM-cut procedure are disjoint. On



(a) (b)

Figure 3.7: Total nodes expanded (a) and total planning time (b) on HBW problems with different heuristics. Instances in each set are sorted by increasing shortest plan length.

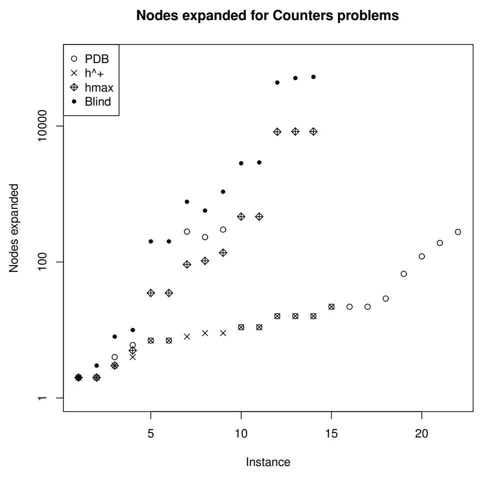
those types of problems, it is possible that the algorithm in Figure 3.6 and the procedure described in Section 2.2.2 find the same collection of landmarks. Both LM-cut and our disjoint landmark heuristic are defined as sum of the costs of landmarks (see Definition 11 and Line 7), they can return the same value.

However, both LM-cut and our algorithm use arbitrary tie-breaking, so in practice, the sets of landmarks returned by the two heuristics may by different (and therefore the heuristic cost estimates might be different). In our case, the NewLandmark procedure extends the set of actions A by adding actions from $\overline{A}$ in an arbitrary order. LM-cut arbitrarily chooses which action preconditions to discard when constructing the justification graph (and there is often high variability in quality of LM-cut with respect to this choice).

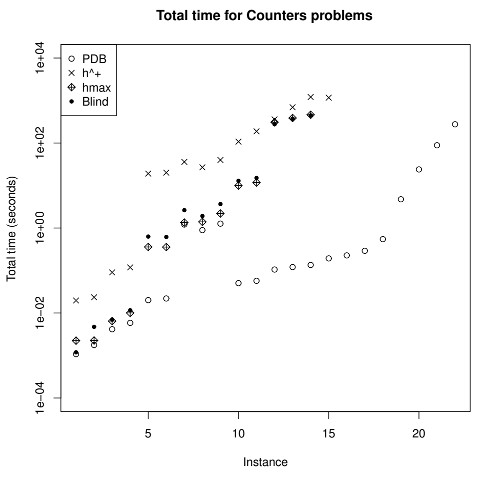
3.6 Experiments

Here we will present the experiments comparing the performance of the heuristics: blind (i.e. returning h value of 0 for every state), $h^{\max}$, h^{+} and PDB. The pattern in the PDB heuristic was selected according to the method described in Haslum et al. [72]. The sets of problems are the following:

- The set of HBW problems consisted of 69 instances, with 4 to 7 block

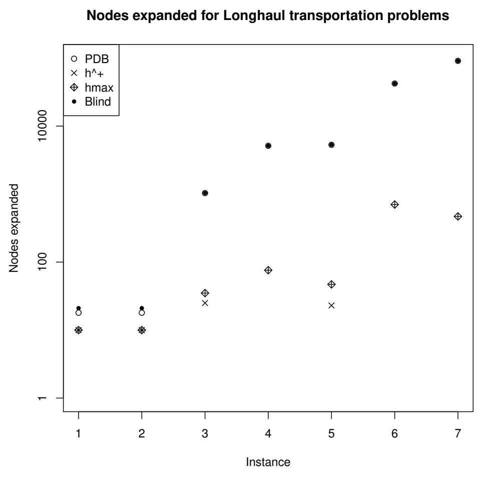


(a)

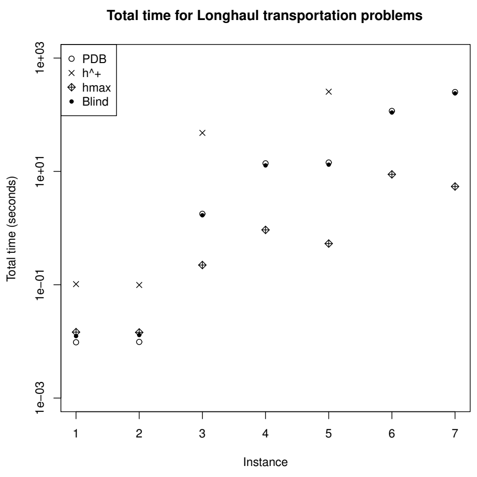


(b)

Figure 3.8: Total nodes expanded (a) and total planning time (b) on Counters problems with different heuristics. Instances in each set are sorted by increasing shortest plan length.



(a)



(b)

Figure 3.9: Total nodes expanded (a) and total planning time (b) on Counters problems with different heuristics, sorted by increasing shortest plan length. Note that only the instances solved with blind and PDBs are presented on this graph. $h^{\max}$ is capable of solving a much greater number of instances.

Domain	Number of problems	Blind	$h^{\max}$	h^{+}	PDB
HBW	69	69	69	58	69
Counters	22	14	14	15	22
Long-haul transportation	62	7	58	4	7

Figure 3.10: Number of instances solved in each domain and with each configuration within a given time limit.

and 3 to 5 cylinders, created using Slaney and Thiébaux [130] Blocks World generator.

- A set of 22 Counters problems, with between 2 and 24 counters – 4 with 2 counters (3 with random initial values), 4 with 4 counters (3 with random initial values), 2 with 5 counters, 3 with 6 counters 3 with 7 counters and one problem each with 8, 12, 16, 20 and 24 counters. In all problems with non-random initial state, all counters were set to zero at start.

- A set of 62 Long-haul transportation instances, based on data from a Queensland-based transportation company [94]. Each of the problems involved 6 locations (with different demands for each type of good at each location). The fleet consisted of 4 ambient temperature trucks and 4 refrigerated trucks.[5]

The CPU time was limited to 1800 seconds for the Counters, 3600 seconds for the HBW instances and 600 seconds for the Longhaul Transportation domain. The results are shown in Figures 3.7, Figure 3.8 and 3.9. The numbers of instances solved (within the time limit) using each of the configurations and in each of the domains are presented in Figure 3.10.

From the results we can observe that the usefulness of the heuristics varies by domain. In HBW and Counters domains, PDB and h^{+} perform best in terms of node expansions (blind search performs the worst, as it is to be expected). The PDB is, however, convincingly cheaper in terms of time (h^{+} is the slowest). $h^{\max}$ is both much slower and worse in terms of node expansions than PDB. In contrast, in the Long-haul transportation domain, $h^{\max}$ is the only heuristic that solves a significant portion of instances, while PDB performs nearly identical to Blind in terms of nodes expanded and total time (note, however, that the number of instances solved with those two heuristics is very small).

[5]While the data consisted of 364 days in total, we only used the instances that we knew in advance could be solved with this fleet of vehicles.

3.7 Future work

One of the properties of our framework is that the constraints place restrictions on each state in the plan trajectory independently. As we move from one state to another, the values of the secondary variables are discarded and then recomputed regardless of what their values were in the state before. This means that we cannot express problems which may require persistence of secondary variables – for example trajectories in space or slow moving network flows. Adapting our framework to allow for modelling of these types of domains is the subject of future work.

Chapter 4

State-dependent action costs

In the PSR problem, it is desirable to maximise the power supplied to consumers in the time period between starting the restoration operation and reaching the final configuration. Assuming that the shortest plan minimises the unsupplied load over time, the simplest solution is to assign all actions equal and constant costs, and the cheapest plan becomes the one involving the fewest switching operations. There is, however, no guarantee that such assumption holds. The actual cost that we wish to minimise is the sum of *un*supplied loads at each time step. In our framework, this can be expressed as a sum of action costs, with the costs being dependent on the state in which the action is taken. We assume that all the action have equal durations and that the duration between all action applications are equal. The cost of taking an action becomes the sum of unsupplied loads in the state the action is applied to. Previous work that dealt with state-dependent action costs (SDAC) involved only problems without the secondary state constraints. As secondary variables were absent, action costs were functions only over primary variables. In our case, the status of each load is represented by a secondary variable, “fed”, so our action cost needs to be a function of extended state.

This chapter is organised as follows. In Section 4.1, we will go over related work. Section 4.2 will present cost functions that we will use and discuss the difficulties associated with calculating the state-dependent action costs. Section 4.3 will present an adaptation of h^+ heuristic to this setting. Results of the PSR experiments will be given in Section 4.4 and, finally, in Section 4.5 we will discuss future work.

4.1 Related work

Formulating state-dependent action costs in unrestricted numeric PDDL [49] is straightforward, since problem metric can be any fluent expression and actions can have arbitrarily complex and conditional effects on numeric fluents. Another way to express state-dependent action costs is through PDDL3's preferences in action preconditions [62] which incur a penalty (cost) when applying an action in a state in which the preference is unsatisfied. However, optimal planning with state-dependent action costs has received relatively little attention. The optimal MIPS-BDD planner handles an expressive fragment of PDDL3's preferences, but not preferences in action preconditions [42]. The LTOP planner [137] extends optimal planning to complex objectives, including action costs that are a function of action's duration, but not of state the action is applied in. Neither of these approaches are actually applicable for the problem we are discussing here.

Recently, Geißer et al. [58–60] presented ways of adapting heuristics for planning with state-dependent costs, namely additive heuristic h^{add} [17] (which is inadmissible and therefore cannot be used in optimal planning) and an (admissible) abstraction heuristic. Central to their approach is representing cost functions as edge-valued multi-valued decision diagrams (EVMDDs) [25,101] and using them to compute costs in relaxed and abstract states. Abstract cost for a given abstract state s^A and an action a is defined as minimal of applying a over all the states abstracted to s^A. The authors show that, for Cartesian abstractions [4], this value can be efficiently computed using EVMDDs. They then extend counterexample guided abstraction refinement (CEGAR, which is an established method for deriving Cartesian abstractions [27,128]) to this setting. CEGAR, in its original formulation iteratively refines the abstraction until a specific criterion is met, or until an optimal abstract plan is *concretiseable* (i.e. it can be applied to the original problem). In the former case, the resulting abstraction is used to compute PDB heuristic. In the later case, the computed plan is also an optimal plan for the original problem (since cost of an optimal plan in an abstract state space is a lower bound on the optimal cost in the concrete state space). Geißer et al. generalise the CEGAR algorithm to SDAC planning by adding detection of *mismatch flaws*, which occur whenever applying an action has different cost in the abstract state space than in the original state space.

4.2 Formalism

In Section 3.2.6 we mentioned that having state constraints in a planning problem opens the possibility for an action cost to be a function of an extended state. Our interest in state-dependent action costs was motivated by the need to more meaningfully evaluate the plans for the PSR problems. An action should be more expensive if it taken in a state in which fewer buses are fed. Therefore, the the cost of the action depends on the "fed" statuses of the buses, which are secondary variables. Thus, we require formalism for expressing the action cost as a function of the extended state.

Definition 26. *We define a* conditional cost *as a pair $\langle \varphi, c \rangle$ where φ is a partitioned condition and c is a positive constant cost.*

The primary part of φ is denoted φ_P and the secondary part φ_S. Each action a has a set of conditional costs associated with it:

$$\mathit{cost}(a) = \{\langle \varphi_i, c_i \rangle\}_{i=1\ldots k}$$

If a has an unconditional cost, we denote that $c_{\mathit{unconditional}}$ with the corresponding term $\langle \varphi_\emptyset, c_{\mathit{unconditional}} \rangle$. Given an extended state s_{extended} and a set of conditional costs, the cost of applying an action a in s_{extended} is

$$\begin{aligned} \mathit{cost}(a, s_{\mathit{extend}}) \;=\; c_{\mathit{uncond}} + \sum \; & \{c_i | \langle \varphi_i, c_i \rangle \in \mathit{cost}(a), \varphi_{i,P} \text{ holds in } s_{\mathit{extend}} \\ & \text{and } s_{\mathit{extend}} \text{ satisfies } \operatorname{active}(\varphi_{i,S}, s_{\mathit{extend}})\} \end{aligned}$$

Given that there are possibly multiple, or an infinite number of extended states corresponding to a single assignment of primary variables, we need to decide which one to use to evaluate the action cost. In Section 3.2.6, we stated that in some domains the planner may be allowed to chose the *cheapest* extended state such that any secondary preconditions of an action, $\text{pre}_S(a)$, are satisfied. Therefore we define the action cost in the following way:

Definition 27. *$\mathit{cost}(a, s)$ is the cost of the cheapest extended state that is consistent with* $\text{active}(\text{pre}_S(a) \cup C_{\mathsf{inv}}, s)$.

Proposition 8. *Given an action a and a state s, the problem of finding $\mathit{cost}(a, s)$ is NP-hard.*

This is because the values of the secondary variables must be chosen such that fewest conditions φ_i, weighted by cost c_i, are satisfied. Given a set of conditional costs and a (reduced) state s, it is straightforward to determine whether the primary part of each of the conditions holds (if it doesn't the associated cost does not apply). Among those for which the primary part of the condition holds, we then have to look at those for which active($\varphi_{i,S}, s$) is not an empty set (if the primary part holds in s and the set of active constraints in s is an empty set, then φ_i holds in s and c_i has to be added to the action cost). Given active($\varphi_{i,S} \cup C_{\mathsf{inv}}, s$) and associated costs c_i, the associated costs count towards the action costs iff the assignment of secondary variables satisfies active($\varphi_{i,S} \cup C_{\mathsf{inv}}, s$). The cheapest extended state is therefore the one that satisfies the fewest secondary conditions (or *soft constraints*) weighted by associated costs (the active invariant constraints in the state are the *hard constraints*). As in Section 3.5.1, we can show that this problem is NP-hard by reducing a problem of satisfying a set of linear inequalities with 0-1 integer variables problem (or zero-one integer linear program without optimisation), which we know is NP-hard [90], to this kind of problem.

Proposition 9. *Given two sets of linear inequalities $Ax \leq b$ (hard constraints) and $Cx \leq d$ (soft constraints), where A and C are a $m \times n$ matrices, b and d are an m-vectors and x is a n-vector, such that $x \in \{0,1\}^n$, finding x such that the number of soft constraints that are satisfied is maximum*[1] *is NP-hard.*

Proof. A zero-one integer linear program without optimisation consists of a set of constraints $Ax \leq b$, with A being a $m \times n$ matrix, b is an m-vector and x is a n-vector of variables. Domains of all the variables $x_i \in x$ are $\{0, 1\}$. This problem is known to be NP-hard [90] and we can rewrite it in the formulation given above as: (i) $Ax \leq b$ are the hard constraints of the reduced problem, (ii) a constraint $0 \leq x_i \leq 1$ for every variable is added to this hard constraints and (iii) $x_i \leq 0$ and $x_i \geq 1$ for every variable are the soft constraints of the reduced problem. Constraints in (ii) and (iii) ensure that the assignment of each x_i is either 0 or 1, as required by the original problem. Solving the reduced problem means finding the maximum number of constraints in (iii) that can be satisfied (we assign equal cost to all of the constraints, so the objective becomes finding an assignment that satisfies the most of them). If the original problem has a solution, then one of the constraints in (iii) for each x_i can be satisfied: this is because if the original problem is consistent each $x_i \in x$ is either 1 or 0 and therefore satisfies exactly one of the two inequalities associated with it in (iii). If

[1]Note that finding a minimum number of inequalities that can be satisfied is equivalent to multiplying each inequality in $Cx \leq d$ by -1 (flipping their signs) and finding the maximum number that can be satisfied.

the the original problem is inconsistent, then there is no such assignment to variables in x, meaning that the maximum number soft constraints that can be satisfied is less than half: if it were consistent, each variable would be either zero or one, therefore satisfying half of the soft constraints. (we can never satisfy more than half soft constraints, because both soft constraints in (iii) for the same variable cannot hold at the same time). Solving this problem therefore tells us whether the original problem is consistent or not. Solving the converted problem is therefore NP-hard. □

Luckily, there are cases where calculating the state dependent action cost is easier. If the conditions in each of the conditional costs have only the primary part (i.e. secondary part is an empty set of switched constraints), we have the problem that is the same as classical planning with state dependent action costs. Evaluating the the cost of an action in a given state simply becomes evaluating the primary partial variable assignment. As an example where the cost depends on the extended state, we might have cases where the portion of the cost that depends on the secondary variables can be expressed as a linear function only over the secondary variables. Here finding the cheapest extended state becomes a linear programming problem. I.e. we might have the portion of the cost that is a function over only the secondary variables $c_{secondary} = \sum_i a_i v_i$, where a_i are constants and v_i are secondary variables. In this case we have a linear programming problem with active$(\text{pre}_S(a) \cup C_{\mathsf{inv}}, s)$ as constraints and $c_{secondary}$ as the objective function. The cost of applying a in s becomes $cost(a, s_{extended}) = c_{unconditional} + c_{secondary} + \sum_i \{c_i | \langle\langle \varphi_{i,P}, \emptyset\rangle, c_i\rangle \in cost(a)\}$ (the final element are the conditional costs in which the conditions have only the primary part). The linear programming problem is known to be solvable in polynomial time [93].

4.2.1 Cases where all the extended states have the same cost

We will, however, focus on simpler cases in which the problem of finding the cheapest extended state can be avoided – here we will discuss the domains where all of the extended states have the same cost, yet the cost still depends on the values of a subset of secondary variables. This is true when the subset of secondary variables that affects the conditional costs is assigned unique values given an assignment of primary variables. The state constraints are, however, still involved in finding this cost, as the values of the secondary variables are determined using the state constraints. Even if a single state corresponds to a (potentially infinite) number of extended states, as long as all the secondary variables that the conditional cost may depend on are uniquely defined given an assignment of primary variables, calculating their costs and incorporating it in search is straightforward.

A condition $\langle \varphi_P, \varphi_S \rangle$ is uniquely defined in a given state if all the secondary variables that appear in the switched constraints in φ_S have unique values in all of the states. We can then evaluate each condition one by one by (i) evaluating φ_P (ii) determining whether active($\varphi_S \cup \text{pre}_S(a) \cup C_{\mathsf{inv}}, s$) is satisfiable. If both the primary and the secondary part of the condition are true, the associated cost is added to total cost. Finding the state-dependent action cost can in this case be done in polynomial time, i.e. the time necessary to evaluate the conditions (which involves solving the linear programming problem) and to sum up the costs.

While this case may seem restricted, it still allows us to model interesting and practically useful problems, such as PSR. In PSR, the secondary variables affecting the costs are "fed" statuses of the buses and the position of the switches uniquely determines whether each of the buses is supplied with power or not. The conditional costs are the same for every action, $\mathit{cost}(a, s) = \{\langle\langle p_\emptyset, p_\emptyset \rightarrow f_i = 0\rangle, \bar{l}_i\rangle | i \in \mathcal{B}\}$. That is, if the bus is not fed, $f_i = 0$, the cost that needs to be added to the cost of the action is the load associated with the bus, $\bar{l}_i$. Each action also has a constant cost $c_0 = 1$, in order to minimise the total number of switching operations.

4.3 Computing heuristics

Extending h^+ to account for state-dependent action costs is more challenging. We will discuss only the case where all the extended states have the same cost.

First, we will define a *negation* of a partitioned condition (denoted $\neg\varphi$ for a partitioned condition φ). The negation of a partitioned condition holds in every state in which the original partitioned condition does *not* hold, which is the case whenever its primary part φ_P doesn't hold or active($\varphi_S \cup \text{pre}_S(a) \cup C_{\mathsf{inv}}, s$) is unsatisfiable. $\neg\varphi$ does not hold in every state in which the φ holds.

The way the state-dependent costs are formulated is essentially as conditional action effects (see Haslum [69]). Therefore, it is possible to apply the same kind of problem transformation that is used to compile the conditional effects to reduce the problem to the one that has constant action costs. That is, replace each action a with a number of copies of an action, one a_X corresponding to each subset $X \subseteq \mathit{cost}(a)$. Each copy's cost is constant $\mathit{cost}(a_X) = c_0 + \sum\{c_i | \langle \varphi_i, c_i \rangle \in X\}$, i.e. the sum of the costs in the subset X. Precondition of the copy of an action becomes

$pre(a_X) = pre(a) \wedge \{\neg\varphi_i | (\varphi_i, c_i) \in cost(a) - X\}$, ensuring that the action is applicable only in states where the conditional costs not in X apply.

The problem with this approach is exponential blow-up. In one of our PSR problems, there are 45 loads, meaning that each action would end up having 2^{45} copies. To avoid this, we take an approach similar to optimal relaxed planning with conditional effects [69]. The procedure works as follows:

1. We assume that every action will be applied in least-cost state, therefore making its cost constant. We compute an optimal relaxed plan under this assumption, by using the algorithm from Section 3.5.4 (Figure 3.5). This gives us a set of actions in the relaxed plan A. The cost of the relaxed plan is the sum of costs of actions in A.

2. We use systematic branch-and-bound search to try to sequence the actions in A so that the assumed cost is actually achieved. If this is possible we are done.

3. If such sequencing is not possible, we record which of the conditional costs got triggered for each action (therefore preventing us from achieving the minimum cost). This gives us a subset of actions A_c and a corresponding list of conditional costs for each action. For each of the actions a in this set:

   (a) We select one of the triggered conditional costs of a, $\langle\varphi_i, c_i\rangle$.

   (b) We split a into two copies.

   (c) One copy, a^0 gets the added precondition $\neg\varphi_i$ (corresponding to any choice of X with $\langle\varphi_i, c_i\rangle \notin X$).

   (d) The second copy a^1 has c_i added to its unconditional cost (corresponding to any choice of X with $\langle\varphi_i, c_i\rangle \in X$).

   (e) We remove $\langle\varphi_i, c_i\rangle$ from the conditional costs of both a^0 and a^1.

   We return to the Step 1 with the modified action set.

The process is repeated until the step 2 succeeds.

4.4 Experiments

We tested the procedure described in the previous section on 45 single-fault PSR problems. The network consisted of 7 generators, 45 power-lines and 45 loads, all of different, predefined capacities. The location of the fault was

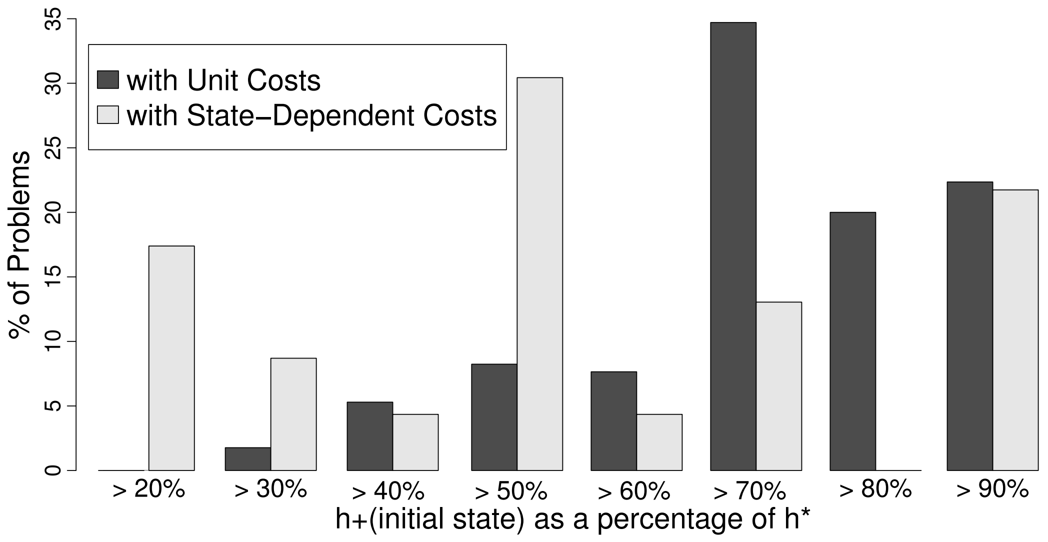


Figure 4.1: Accuracy of the relaxed plan heuristic with state-dependent action costs, and with unit cost actions. The accuracy in the unit-cost case here is different from that in Figure 6.3(a), because the problem set is smaller.

different in each problem and the goal consisted of a subset of loads that have to be fed.

The iterated relaxed plan construction is very effective at reducing the number of copies of actions: For the initial states of these problems the average number of iterations was 3.62 and the average number of action copies was less than ten. In contrast, full compilation would have created 2^{45} copies of each action. However, this heuristic is not time efficient. The time to evaluate a single state is, on average 6.85 times higher than in the unit-cost case. This reflects both the repeated h^+ computation and the overload of sequencing the action set. The heuristic becomes less accurate than it is with constant costs, as shown in Figure 4.1. Consequently, blind search solves more of these problems than search with the h^+ heuristic.

4.5 Future work

Future work should focus on developing computationally cheaper heuristics for this setting. One possible solution could be to build on the approach by Geißer et al. [60], described earlier (Section 4.1). However, their method for calculating cost of applying an action in the abstract state cannot be easily applied to our case: to find abstract cost values, they encode a cost function for each action as an EVMDD, whose nodes are (primary) variables, and perform local minimisation over the edges (which correspond to values of the primary variables) of the EVMDD. Given an abstract state, it is clear which values are possible for each of variables. In our case, costs may be functions over secondary variables and, for a given abstract state, it is not obvious which values are possible for a given secondary variable. To apply their approach, we need to either adapt EVMDD computation or find an alternative way of efficiently evaluating cost functions in an abstract state.

Chapter 5

Axioms

The type of state constraints that we will explore in this chapter are axioms. Unlike in planning with numeric constraints, here the domains of both primary and secondary variables are finite and discrete. We will show that the addition of axioms can make problems easier to solve as well as easier to model. As we did with numeric state constraints, we will develop a formalism describing the planning problem with axioms and adapt existing admissible heuristics to this setting. We will describe several types of domains where their usage improves performance. Unlike previous work on axioms by other researchers (see Section 2.3.1), we will focus on cost optimal planning.

In Section 5.1 we give reasons for using axioms. Section 5.2 gives a formal definition of axioms in FDR and of a planning problem with axioms, and the relation to axioms in PDDL. Sections 5.3-5.5 describe example domains, which can be grouped as: domains where axioms are used to compute transitive closure, pseudo-adversarial domains and social planning domains. Section 5.6 shows how we compute admissible axiom-aware heuristics. We follow the same approach as we did for the numeric constraints, thus deriving axiom-aware monotonic relaxation based heuristics and abstractions. Section 5.7 presents the experimental results.

5.1 Motivation

In Section 3.4, we noted that numeric constraints can be compiled to classical planning, although with potentially exponential increase in problem size. The same is true of axioms [136]. As we noted earlier (Section 2.3.1), some authors (such as Gazen and Knoblock [53], Garagnani [52], Davidson and Garagnani [36]) held the view that, it might be better to compile them away, rather than to deal with them explicitly. One of their arguments was that this allows for using already existing planners that don't implement axioms.

We will demonstrate that use of axioms on certain classes of domains has two advantages. Firstly, axioms allow us write the domain descriptions in a more concise and readable forms. Thiébaux et al. [136] prove that any compilation scheme results in either a worst-case exponential blow-up in the size of domain description or worst-case exponential blow up in the size of the length of the shortest plan. Secondly, use of axioms can make problems easier to solve, as specifying indirect action effects removes unnecessary choices from the search space of the planner. The second point will be demonstrated by our experiments.

5.2 Formalism

Here we will formally define axioms and planning tasks with this form of global constraints. We will first present a definition given by Helmert [78] whose formalism is compatible with FDR. We will subsequently give the definitions by Thiébaux et al. [136], which is the work that reintroduced axioms into PDDL, and show that this form can be compiled into the one that we are using.

5.2.1 Axioms in FDR

As with numeric state constraints, we make the distinction between primary and secondary variables. Following the terminology used in Chapters 2 and 3, we will refer to variables whose values are determined by action effects as primary variables (or state variables) and variables whose values are determined by state constrains as secondary variables.[1] The definitions here follow Helmert [78] who presents formalism for axioms with discrete domain primary and secondary variables. To make the presentation easier and to simplify the derivation of heuristics,[2] we will only consider domains in which all secondary variables are Boolean. We will, however, explain how to generalise the formalism to finite discrete domain variables.

Each secondary variable has a *default value*, which is retained if we cannot justify assigning it any other value. In all of our domains, the default values

[1]We should note here that other authors use different terms. Thiébaux et al. [136] refer to primary variables as *basic variables* and secondary variables as *derived variables.* Helmert uses the term *fluents* for primary variables and *derived variables* for secondary variables. Confusingly, the same author refers to the union of primary and secondary variables as *state variables.* To be consistent with the terminology used in the rest of our work, we will refer to primary variables as state variables.

[2]Answer set programming uses binary variables, see Section 5.6.2

are always *false*, meaning that everything that cannot be derived as true is assumed to be false – this is referred to as *negation as failure* or *closed world assumption.*

Definition 28. *[Adapted from Helmert [78]]* [3] Axioms *have the form* $v \leftarrow$ *cond where*

- *cond is a partial variable assignment (over both primary and secondary variables), called* condition *or* body *of the axiom and*
- *v is a secondary variable called* affected variable *or* head *of the axiom.*

In any state in which the condition holds, the affected variable is assigned true.

It is required that the secondary variables and the axioms are *stratified.* A set of secondary variables V_S is partitioned into a totally ordered set of *secondary variable layers*, $V_{S,1} \prec \cdots \prec V_{S,k}$. This induces the stratification of the set of axioms $\mathcal{A}$ – i.e. there are as many layers of axioms as there are layers of secondary variables and the layer an axiom is in corresponds to the layer the secondary variable in the head of the axiom is in, $\mathcal{A}_1 \prec \cdots \prec \mathcal{A}_k$ (i.e. if the affected is in the nth secondary variable layer, than the axiom is in the nth axiom layer). Within the same layer, each affected variable must appear with the unique value in all axiom heads and bodies. That is, within the same layer, we cannot have axioms with the different derived values for the same variable, and if a variable appears in the head of an axiom, it may not appear with a different value in in its body.

When all of the secondary variables are Boolean and all of the default values are *false*, stratification means that the secondary variables belonging to the first layer, $V_{S,1}$, are defined in terms of the primary ones, possibly using *negation* (i.e. allowing for the assignment of *false* in the assignment in the axiom body), or in terms of themselves (allowing for recursion) but without using negation (only allowing *true* to the variable in the axiom body). In the second layer of axioms, variables in heads of the axioms belong to $V_{S,2}$ layer, while the body consists of variables in the $V_{S,1}$ layer (possibly using negation) and the primary variables (again, possibly using negation), or in terms of themselves, but without using negation. Each subsequent layer is then built using the secondary variables in the lower levels, possibly using negation or

[3] As Helmert allows for non-binary variables, axioms in his formalism are triples $\langle cond, v, d\rangle$, where *cond* and v are the same as in our definition and $d \in \mathcal{D}(v)$ is called the *derived value* for v. In any state in which the condition holds, the affected variable is assigned the value d.

```
1: procedure EVALUATE AXIOMS($\mathcal{A}_1, \dots, \mathcal{A}_k, s$)
2:   for variable $v$ do
3:     if $v$ is a primary variable then
4:       $s'[v] = s[v]$
5:     else if $v$ is a secondary variable then
6:       $s'[v] = \mathit{false}$
7:   for $i \in \{1, \dots, k\}$ do
8:     while there exists an axiom $v \leftarrow \mathit{cond} \in \mathcal{A}_i$ with $s'[\mathit{cond}]$ and $s'[v] = \mathit{false}$ do
9:       Choose such an axiom $v \leftarrow \mathit{cond}$
10:      $s'[v] = \mathit{true}$
```

Figure 5.1: Algorithm for evaluating a layered set of axioms $\mathcal{A}_1, \dots, \mathcal{A}_k$, given an assignment of primary variables s. Adapted from Helmert [78]. The differences between the procedure presented here and in Helmert's work is that, as secondary variables are not limited to binary variables, (i) the value initially assigned to the variable is not neccesarily *false* (but its *default value*) and (ii) the axiom has the form $\langle \mathit{cond}, v, d \rangle$, where d is any value from $\mathcal{D}(v)$ other than the default value. The Line 10 assigns d to $s'[v]$.

in terms of themselves (without negation). Primary variables may be used freely (possibly negated) in any layer. When no secondary variable occurs with the value *false* in the body of any axiom, there is only one stratum. A number of researchers have considered this special case, namely Gazen and Knoblock [53], Garagnani [52] and Davidson and Garagnani [36].

Stratification guarantees that this procedure terminates and produces the deterministic result for a given state. That is, starting with the lowest layer, evaluating axioms in any order within that layer, only moving to the next layer after all of the axioms within that layer have been evaluated, and working upwards until all of the axioms in the highest layer have been evaluated always gives the same assignment [136]. This procedure is presented in in Figure 5.1.

5.2.2 Planning problem with axioms

The definition of a planning problem with axioms is very similar to the definition of the planning problem with numeric state constraints, with several notable differences. Secondary variables have discrete finite domains instead of real number domains and switched constraints are replaced by axioms. Goals and action preconditions are partial variable assignments over both primary and secondary variables. We can think of action preconditions and goals as partitioned conditions (Definition 18) $\langle \varphi_P, \varphi_S \rangle$ in which the secondary part φ_S is a partial variable assignment over the secondary variables. Note, however, that the action effects are still partial variable assignments only over the primary variables. Likewise states (including the initial state) are still defined as variable assignments only over the primary variables. The definition of a planning problem with axioms is therefore:

Definition 29. *A* planning problem with axioms *is defined by a tuple* $\Pi = \langle V_P, V_S, C, A, s_0, G, cost \rangle$ *where*

- V_P *is a set of primary variables, with each* $v_P \in V_P$ *being associated with a finite domain* $\mathcal{D}(v_P)$*. A* state *assigns each primary variable a value from its domain.*
- s_0 *is an initial state.*
- V_S *is a set of secondary variables, with each variable being associated with a Boolean domain,* $\{true, false\}$*.*
- C *is a set of layered axioms.*
- G*, the* goal*, is a partial variable assignment over both primary and secondary variables.*
- A *is a finite set of actions. Each action* $a \in A$ *is a pair* $\langle \mathrm{pre}(a), \mathrm{eff}(a) \rangle$*.*
  - $\mathrm{pre}(a)$*, or* precondition*, is a partial variable assignment over both primary and secondary variables.*
  - $\mathrm{eff}(a)$*, or* effects*, is a partial variable assignment only over the primary variables.*
- $cost(a, s)$ *is a cost function with* a *being an action and* s *an extended state.*

Same as with planning with numeric constraints, we have a concept of an *extended state.*[4] While a state (or a reduced state) is an assignment over all primary variables, an extended state is an assignment over all primary and all secondary variables. As already stated, stratification ensures that there is a unique extended state associated with every state.

The definitions of action applicability, plans and plan costs are exactly the same as in FDR planning (see Section 2.1.1).

[4]The term was, in fact, originally introduced by Helmert for planning with axioms.

5.2.3 Axioms in PDDL

We will now give a very brief overview of axioms in the form they were (re-)introduced to PDDL by Thiebaux et al [136]. We will show that PDDL axioms can be converted to the FDR formalism given earlier. Helmert [78] uses the FDR formalism to show how to translate PDDL tasks (with axioms) to a grounded representation that uses finite domain variables. That compilation procedure is used in his Fast Downward planning system [78], which is a planner that we use as well. While PDDL is a restricted first-order formalism, planning systems such as Fast Downward compile the problem specification into a propositional representation by grounding predicates, operators and goal specifications.

In PDDL with axioms distinction is made between *basic predicates* (also called *fluent* predicates) and *derived predicates*, with $\mathbb{B}$ and $\mathbb{D}$ being used to denote sets of basic and derived predicate symbols, respectively (with those two sets being disjoint $\mathbb{B} \cap \mathbb{D} = \emptyset$).

The notation used in PDDL is as follows:

Definition 30. *PDDL axioms are of the form* $(:\texttt{derived}(d?\vec{x})(f?\vec{x}))$*, where*

- $d \in \mathbb{D}$ *is a derived predicate and*
- f *is a first-order formula built using both basic and derived predicates and whose free variables are those in vector* $\vec{x}$*.*

Here $(d?\vec{x})$ is the *consequent* (corresponds to the head in FDR) of the axiom and $(f?\vec{x})$ is the *antecedent* (corresponds to the body). of the axiom. Intuitively, $(:\texttt{derived}(d?\vec{x})(f?\vec{x}))$ means that whenever $(f?\vec{x}))$ is true, we should *derive* that $(d?\vec{x})$ is true in the same state (note that, unlike in FDR, we are here dealing with Boolean variables). As with FDR axioms, everything that cannot be derived as true is assumed to be false. For this reason, consequents are restricted to the contain only positive literals [136]. The distinction between primary and secondary variables in FDR arises from this division of predicates into basic and derived.

Here we will show that PDDL axioms can be compiled to the form presented in Section 5.2.1. While grounded PDDL allows for an arbitrary formula in the body of an axiom, our FDR representation requires the body to be a partial variable assignment. The arbitrary formula can be rewritten as a partial variable assignment with polynomial increase in size. This is done by rewriting it in a conjunctive normal form (CNF) and introducing a new variable corresponding to each clause. The formula, converted to CNF is $q \leftarrow \bigwedge_i \bigvee_j (\neg) x_{i,j}$. We introduce a new variable corresponding to each clause (i.e. p_i for clause i) and a set of axioms with one axiom corresponding to each term in the clause and rewrite the original axiom q as a partial variable assignment over the newly introduced variables.

$$\begin{aligned} &p_1 \leftarrow (\neg)x_{1,1} \\ &\vdots \\ &p_1 \leftarrow (\neg)x_{1,j} \end{aligned}$$

Finally, the original axiom q is rewritten as a partial variable assignment over the newly introduced variables $q \leftarrow \bigwedge_i p_i$.

5.2.4 Comparison with numeric state constraints

Here we will give a few remarks on the differences between the switched constraints and axioms. The similarities are obvious – in both formalisms primary variables function as in classical planning (in both cases primary variables have discrete and finite domains, respecting the Assumption 1 of classical planning), while state constraints are used to evaluate secondary variables. In both cases, the problem can be compiled to classical planning (see Section 2.3.1 for axioms and Section 3.4 for numeric constraints), although this leads to an exponential increase in problem size.

One of the differences is in relation between reduced states and extended states. An important consequence of the layering property of axioms is that, given a reduced state, there is guaranteed to exists a unique assignment to secondary variables that satisfies the constraints (so there is one unique extended state corresponding to assignment of primary variables). In contrast, in planning with numeric state constraints, there is no guarantee that there are any extended states corresponding to a given assignment of primary variables (in which case such an assignment is an invalid state). On the other hand, for a given reduced state, we could have a single extended state, a finite number of extended states or an infinite number of extended states.

5.3 Modelling with Axioms

All planning problems that can be formulated with axioms can be formulated without using them. We can however, show that in certain classes of domains, axioms not only make problems easier and more compact to formulate, but also easier to solve. The difference between the formulation with and without axioms will be shown in Section 5.7.

We will discuss three groups of domains where the use of axioms can be beneficial:

- The domains where we reason about *transitive closure* properties (i.e. exsistence of paths in a graph or reachability of flows). This group was identified by Thiébaux et al. [136] in their work on axioms.

- Pseudo-adversarial domains – the domains in which we are required to reason about actions of an opponent that follows some complex, yet deterministic rules.

- Related to the previous group are the social and multi-agent planning domains.

5.3.1 Min-Cut

This is an example of a domain where we use axioms to compute transitive closure relations. Consider an undirected graph, with a source node s and a target node t. K roadblocks are located on the edges of the graph and can be moved between adjacent edges. More than one roadblock can occupy the same edge and one roadblock can pass another on the edge. The goal is to move the roadblocks so that there is no unblocked path from the source to the target node. In other words, a goal state identifies an $s - t$-cut of size $\leq K$. Hence, we call it the Min-Cut domain. Figure 5.2 shows a small instance, with roadblocks A and B.

The primary variables are the locations of roadblocks, at_l, and their domain is the set of edges. In the PDDL encoding of the domain this is expressed with a function that maps the block to the edge. The derived predicates are: $\mathsf{blocked}(i, j)$, $\mathsf{reachable}(i)$ and $\mathsf{isolated}(i)$. The complete PDDL encoding can be found in the appendix. Translated to finite domain representation, the secondary variables and their defining axioms are:

$$\begin{array}{lcl}
\mathsf{blocked}_{i,j} & \leftarrow \mathsf{at}_l = e_{i,j} & l = 1, \ldots, K \\
\mathsf{reachable}_i & \leftarrow \mathsf{source}_i = \mathit{true} & \text{(source node)} \\
\mathsf{reachable}_i & \leftarrow \mathsf{reachable}_j = \mathit{true}, \mathsf{blocked}_{i,j} = \mathit{false} & j \in N(i) \\
\mathsf{isolated}_i & \leftarrow \mathsf{reachable}_i = \mathit{false} &
\end{array}$$

In the third axiom $N(i)$ is the set of neighboring nodes of i, meaning that this line represents a set of axioms with one for each of the neighbors. Note that $\mathsf{reachable}_i$ means that i is reachable from s by an unblocked path. The action, $\mathsf{move}_{l,e_{i,j},e_{j,k}}$ moves the block l, changing its location at_l from $e_{i,j}$ to $e_{j,k}$.

In the state in Figure 5.2, $\mathsf{at}_{\mathbf{A}} = e_{1,5}$ and $\mathsf{at}_{\mathbf{B}} = e_{3,6}$, so we can derive $\mathsf{blocked}_{1,2} = \mathit{false}$ and $\mathsf{blocked}_{2,6} = \mathit{false}$ and from this $\mathsf{reachable}_1 = \mathit{true}$,

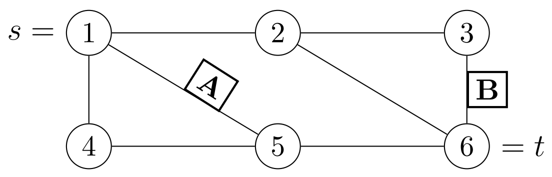


Figure 5.2: An example problem from the Min-Cut domain.

$\mathsf{reachable}_2 = \mathit{true}$ and $\mathsf{reachable}_6 = \mathit{true}$. To achieve the goal, the two road-blocks must move to edges $e_{1,2}$ and $e_{5,6}$.

5.3.2 Sokoban

Sokoban was originally created as a single-player computer game developed by Hiroyuki Imabayashi and published by *Thinking Rabbit* in 1982. It was later formulated as a planning problem and used in International Planning Competitions (IPC). As in the Min-Cut domain, we are using axioms to express the transitive closure relation.

An example Sokoban problem is depicted in Figure 5.3. It features a man who can move around a maze and push stones, one at a time. The goal is to push "stones"[5] to the designated goal positions. The goal positions are shown as red circles here. The stones that are not in the goal locations are shown in lighter colour, while the ones already at the goal locations are darkened. The stones are identical and interchangeable – i.e. any stone can be pushed to any of the goal positions. The man can move only through the empty spaces (i.e. he cannot move through the stones or the maze walls).

The cost function that we are using assigns cost of one to all pushes and zero to all move actions. That is, we are minimising the number of pushes; the moves in between the pushes don't count, as long as he is able to reach the square that is next to the stone that is to be pushed. The problem is usually modelled by having stepwise (non-pushing) moves of the man as actions with zero cost. This forces the planner to make an irrelevant choice of the exact path the man takes between each push action, increasing the size of the state space and plan length.

Reachability of a particular location is straightforward to express as a recursively derived property, given the current arrangement of stones. Therefore, using axioms, it is possible to formulate the problem with pushing

[5]They are depicted in Figure 5.3 as boxes, but usually referred to as "stones" in literature. The Japanese word "sokoban" means warehouse keeper, so boxes are what they were most likely meant to represent in the original game.

actions only, allowing the man to "jump" from one stone to the next as long as the path between them exists.

The complete PDDL formulation of the domain is given in the Appendix B. Here we will show some of the axioms in PDDL and the translation to finite domain representation. The axioms are used to determine whether stones are at the goal locations. $\textsf{is-goal}(l)$ and $\textsf{at}_(s,l)$ where l is a location and s is a stone. Firstly, we use them in an axiom defining the derived predicate $\textsf{at-goal}(s)$, which tells us whether a stone s is at any of the goal locations:

```
    (:derived (at-goal ?s - stone)
              (exists (?l - location) (and (is-goal ?l)
                                            (at ?s ?l))))
```

In the translation, the derived predicates become secondary variables (again, existential quantifier means that there is corresponding axiom for each l).

$$\textsf{at-goal}_{i,j} \quad \leftarrow \quad \exists l \quad \textsf{is-goal}_l = \mathit{true}, \textsf{at}_{s,l} = \mathit{true}$$

Secondly, the following rules determine whether a location l is *clear* (not blocked by any of the stones s).

```
  (:derived (blocked ?l - location)
            (exists (?s - stone) (at ?s ?l)))
  (:derived (clear ?l - location) (not (blocked ?l)))
```

Translated, they become:

$$\begin{array}{lcll} \textsf{blocked}_l & \leftarrow & \exists s & \textsf{at}_{l,s} = \mathit{true} \\ \textsf{clear}_l & \leftarrow & & \textsf{blocked}_l = \mathit{false} \end{array}$$

Finally, axioms are used to compute whether some location l is *reachable* by a player p.

```
  (:derived (can-reach ?p - player ?l - location)
            (at ?p ?l))

  (:derived (can-reach ?p - player ?l - location)
            (and (clear ?l)
                 (exists (?d - direction ?m - location)
                         (and (MOVE-DIR ?m ?l ?d)
                              (can-reach ?p ?m)))))
```

These translate to:

$$\begin{array}{lll} \textsf{can-reach}_{p,l} & \leftarrow & \textsf{at}_p = l \\ \textsf{can-reach}_{p,l} & \leftarrow \exists m, d & \textsf{clear}_l = true, \textsf{move-dir}_{m,l,d} = true, \textsf{can-reach}_{p,m} = true \end{array}$$

Here m is a location and d is a direction. $\textsf{neighbour}_{m,l,d}$ tells us whether l is the neighbour of m in the direction d.

5.4 Modelling with axioms – Pseudo-adversarial domains

By pseudo-adversarial domains we mean the domains where we are required to reason about the actions of an opponent that follows some complex, yet known and deterministic rules. These are usually game-like planning problems where the planning agent is facing an opponent who is actively trying to disrupt the plan. We call these *pseudo-adversarial* domains.

5.4.1 Controller verification

Ghosh et al. [66] formulate verification of functional requirements of distributed automotive control system as a planning problem. The verification problems can be thought of as examples of pseudo-adversarial problems, in which the planning agent is the *environment* and the adversary is the *control system* – acting according to its specification. The overall purpose of controller verification is to find bugs in the controller.

The environment disturbs the state of the system, while the controller returns the system to one among many safe states. If the control is distributed, then the controller's move typically consists of an orchestrated set of actions across the system components. The components' actions may execute in different sequences due to non-determinism arising out of task scheduling and communication latencies between the components. Ghosh et al. note that it is important to guarantee that the control is correct in spite of this non-determinism. For this reason, they allow the environment to exploit this non-determinism in the controller by choosing the order of applicable control actions. The environment may only take actions when the controller is in a *control stable state*, meaning that no control action is applicable (an environment action may change the state to one that is not control stable). This reflects the fact that the control action application is faster than the pace at which the events in the environment occur. Whenever the environment changes the state of the system, the controller executes one or more applicable control actions to return the system to a (possibly different) control

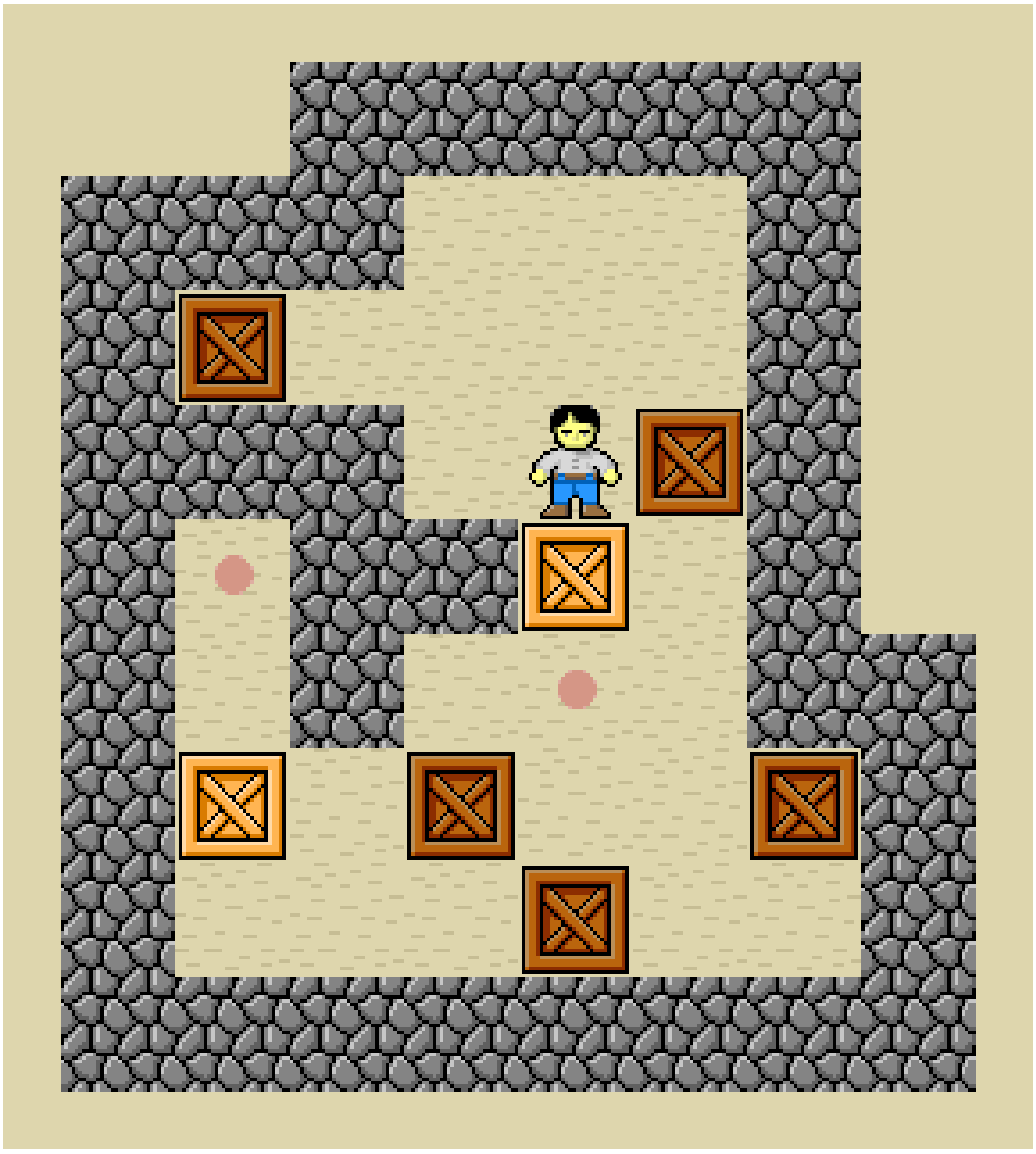


Figure 5.3: Sokoban. From `https://en.wikipedia.org/wiki/Sokoban#/media/File:Sokoban_ani.gif`

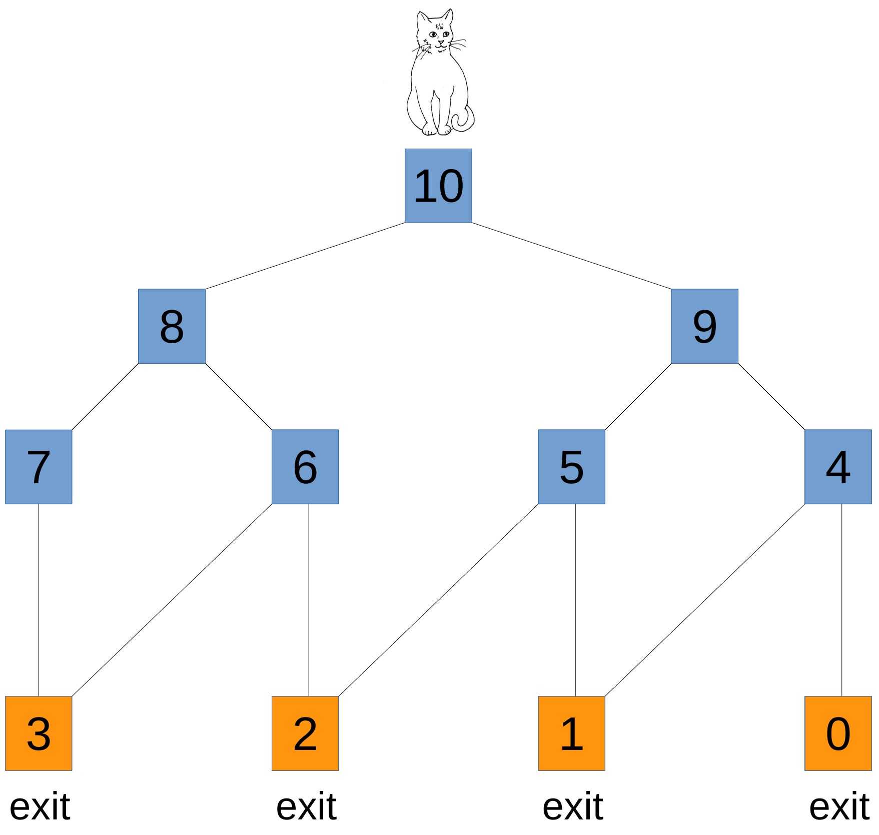


Figure 5.4: Blocker domain.

stable state. A *control fail* is a state in which some of the safety requirements are violated. If the control fail state is reached, it demonstrates the failure of the controller specifications (i.e. controller specifications need to be changed as there is a bug).

The problem can be expressed as a classical planning problem, using STRIPS. The goal of the problem is to reach any of the control fail states – in other words, the existence of a plan for the problem is a counter-example for the safety requirement which is violated in that control fail state. The actions are divided into two disjoint sets, environment actions A_{env} and controller actions A_{ctl}. A Boolean variable en_a is added for each action a which indicates that a may be applicable. A set of conditions $\{\mathsf{en}_a = \mathit{false} \mid a \in A_{ctl}\}$ is added to the precondition of every environment action (that is, the environment takes actions only when no controller action is applicable) and to the goal (no controller action is applicable in the control fail state). A "disabling" action d_l for each literal l that appears in the precondition of some control action a, with $\mathrm{pre}(d_l) = \{l = \mathit{false}\}$ and $\mathrm{eff}(d_l) = \{\mathsf{en}_a = \mathit{false} \mid l \in \mathrm{pre}(a)\}$, is used to mark control actions inapplicable.

Each (environment and control) action that potentially contributes to making $\mathrm{pre}(a)$ true (i.e. that sets any variable to a value required by $\mathrm{pre}(a)$) sets $\mathsf{en}_a = \mathit{true}$, so the plan must include disabling actions before each environment action to verify its applicability. Because the compiled problems are hard for the planners they tried, they also propose an incremental, partial compilation coupled with a plan repair approach.

However, the requirement that no control action is applicable in a control stable state can be easily formulated using axioms:

$$\begin{array}{lcl} \mathsf{stable} & \leftarrow & \{\mathsf{en}_a = \mathit{false} \mid a \in A_{ctl}\} \\ \mathsf{en}_a & \leftarrow & \mathrm{pre}(a) \end{array}$$

Although these rules mirror almost exactly the actions in the compilation by Ghosh et al., making them axioms instead of disabling actions removes the choice of when and which disabling actions to apply from the planner, resulting in a smaller state space and shorter plans. This is similar to the effect that using axioms has in the Sokoban domain – removing unnecessary choices makes the problem easier. The effects of this will be discussed in Section 5.7.2.

Examples

Ghosh et al. present two examples from automotive industry – a car door lock and an adaptive cruise control (ACC) system. We will use their PDDL encodings of these two domains in our experiments (see Section 5.7 for experiments and the Appendix B for the domains). We will describe the door lock system as a simple example. The system is supposed to ensure that all doors are locked when a car attains a pre-calibrated speed. In the example problem, the control system's actions are: (i) if the car switches from low-speed state to high-speed state, the system arms the auto lock procedure; (ii) if the auto-lock procedure is armed and all doors are closed, the the system automatically locks all doors and disarms; (iii) if the remote unlock command is detected, and the doors are locked the system arms the auto-unlock procedure and (iv) if the doors are locked and the auto-unlock procedure is armed, the system unlocks all locked doors and disarms the auto-unlock. The environment's (i.e.the driver's) actions include opening and closing the doors, putting the key into ignition, running the engine, accelerating the car and issuing a remote unlock command. The environment can achieve the goal (i.e. violate the safety requirements) by getting the car into the high speed state and arming the auto-unlock. This shows that the system's action (iii) should have it *not-moving-at-high-speed* added to its preconditions. With preconditions rewritten this way, the environment's goals are not achievable.

The ACC system is a driver assistance feature designed to automatically adjust the vehicle's speed. It operates in two modes: *speed control mode*, in which it maintains the vehicle's speed v_{car} at some chosen v_{ref}, and *time gap mode*, in which it maintains a safe distance (or time gap) between the vehicle and any other nearby vehicles. In the speed control mode the controller's actions are: (i) accelerate if $v_{\text{car}} < v_{\text{ref}}$; (ii) decelerate if $v_{\text{car}} > v_{\text{ref}}$; (iii) maintain constant speed if $v_{\text{car}} = v_{\text{ref}}$ and (iv) switch to the time gap mode if there are other vehicles within the predetermined time gap t_2. In the time gap mode the controller's actions are: (i) accelerate if there is a vehicle directly in front of the car within the time gap t_2 that are going faster than v_{car} and $v_{\text{car}} < v_{\text{ref}}$; (ii) decelerate if there are vehicles within the time gap t_2 that are slower than v_{car}; (iii) maintain speed if the vehicle directly in front is going at the same speed as the v_{car}; (iv) switch to speed control mode if there are no vehicles within t_2 and (v) if there are any vehicles within the pre-fixed time gap t_1, the driver is warned with an audible signal. The environment actions include the driver's actions and the actions of other vehicles (such as switching lanes, accelerating or decelerating). The safety requirements are: (i) if the ACC is engaged and the warning signal is on, the control is also applying negative acceleration and (ii) if the ACC is engaged and there is a vehicle within the t_2 time gap whose speed is slower than v_{car}, then v_{car} must not be increasing.

The authors performed the experiments on the ACC domain problems using a number of model checkers (NuSMV [26] and SPIN [86]), SAT planners and heuristic state-space planners with and without support for derived predicates. They report that, among the tested approaches, the model checker SPIN, and the planners Mp [122], Fast Downward [77] and LAMA [121] show most promise that they would scale to larger problems. The planners, however, generate much shorter plans than SPIN, which makes debugging easier (unlike us, they do not perform optimal planning, so the solutions can be of varying lengths).

5.4.2 Blocker domain

A domain that better illustrates what we mean by term pseudo-adversarial is the blocker domain. Here, we have a game-like problem with two players – the *blocker*, which is a planning agent, and the *cat*, which is an opponent. The game takes place on an undirected graph G on which a subset of nodes are marked as exits. As in Min-Cut domain, we denote neighbourhood of a node n with $N(n)$. The cat's actions consist of moving from a node to one of the neighbouring nodes, always choosing the one that is the closest to the nearest exit node. To ensure deterministic behaviour of the cat, ties are broken by arbitrary numbering of the nodes. The cat moves one node at a time, and in between each of the moves of the cat, the blocker can permanently block any node of the graph that the cat does not currently occupy (the cat cannot move through the nodes that have been blocked). The cat wins if it reaches the exit, while the blocker wins if the cat is blocked from reaching any of the exits.

To formulate the cat's strategy we need to determine which of two neighbouring nodes is closer to an exit:

$$\begin{array}{lll l}
\mathsf{closer}_{n,n'} & \leftarrow & \mathsf{dte}_{n,i} = \mathit{true}, \mathsf{dte}_{n',i} = \mathit{false} & 0 \leq i \leq |G| \\
\mathsf{dte}_{n,0} & \leftarrow & \mathsf{is\text{-}exit}_n = \mathit{true}, \mathsf{blocked}_n = \mathit{false} & \\
\mathsf{dte}_{n,i} & \leftarrow & \mathsf{blocked}_n = \mathit{false}, \mathsf{dte}_{n',i-1} = \mathit{true} & n' \in N(n) \\
\mathsf{dte}_{n,i} & \leftarrow & \mathsf{dte}_{n,i-1} = \mathit{true} &
\end{array}$$

The secondary variable $\mathsf{dte}_{n,i}$ ("distance-to-exit") is true if the shortest distance, along an unblocked path, to an exit from node n is i steps *or less*. Hence, node n is closer to an exit than n' iff $\mathsf{dte}_{n,i}$ is true and $\mathsf{dte}_{n',i}$ is false, for some i. Nodes n and n' are at the same distance ($\mathsf{same}_{n,n'}$ is *true*) iff both $\mathsf{closer}_{n,n'}$ and $\mathsf{closer}_{n',n}$ are false. The shortest distance is bounded by the size of the graph, $|G|$. The variable $\mathsf{order}_{n,n'}$ simply means that n preferred over n' under the arbitrary lexical ordering of the nodes. The cat's reasoning can be expressed as

$$\begin{aligned}
\mathsf{better\text{-}node}_{n,n'} &\leftarrow \mathsf{blocked}_{n'} \\
\mathsf{better\text{-}node}_{n,n'} &\leftarrow \mathsf{prefer}_{n,n'} \\
\mathsf{prefer}_{n,n'} &\leftarrow \mathsf{closer}_{n,n'} = \mathit{true} \\
\mathsf{prefer}_{n,n'} &\leftarrow \mathsf{same}_{n,n'} = \mathit{true}, \mathsf{order}_{n,n'} = \mathit{true}
\end{aligned}$$

Let cat-pos be the primary variable variable representing the cat's current node. The precondition of an action that moves the cat from cat-pos to n is

$$\begin{aligned}
\mathrm{pre}(\mathsf{cat\text{-}move\text{-}to}_n) = \; & n \in N(\mathsf{cat\text{-}pos}), \\
& \{\mathsf{better\text{-}node}_{n,n'} = \mathit{true} | n' \in N(\mathsf{cat\text{-}pos}) \setminus \{n\}\}
\end{aligned}$$

Since $\mathsf{dte}_{n,i}$ is false for all i when there is no unblocked path from n to an exit, the blocker's goal is expressed as

$$\mathsf{trapped} \leftarrow \{\mathsf{dte}_{\mathsf{cat\text{-}pos},i} = \mathit{false} | 0 \leq i \leq |G|\}$$

5.5 Modelling with axioms – Social and multi-agent planning

Multi-agent planning can be similar to pseudo-adversarial planning, except that the other agents are not necessarily opposing the planning agent in accomplishing the goal. Interaction between the agents is more varied and not always adversarial. Agents may attempt to influence the behaviour of other agents, based on the beliefs about others' behaviour in a given situation. In this section we will explain how other authors used axioms in this type of settings.

5.5.1 Chang and Soo

Chang and Soo [24] refer to this as *social planning* and use it for *automated narrative generation.* In Chang and Soo's formulation, characters plan both with their own actions and actions of other characters, but each action taken by another character requires that the character has a motive for achieving the action's effects. Characters use rules to update their mental state according to perception.

The example used in their work is a simplified plot of Shakespear's play Othello. Much of the story can be viewed as an execution of a plan by villain of the play, Iago, who manipulates other characters into carrying out his aims. Iago hates Othello for promoting a younger man named Cassio above him, and decides to make him suffer by making him kill his own wife, Desdemona. To carry out his plan, Iago first asks his wife Emilia to take a handkerchief that Othello gave Desdemona and give it to Iago. Iago then plants the handkerchief in Cassio's residence, expecting that Cassio would pick it up. Othello sees the handkerchief on Cassio and, believing that Cassio received the handkerchief from Desdemona, decides to kill her.

The complete encoding of the problem (in PDDL) is given in the Appendix B. In this particular problem, there are four characters (Iago, Emilia, Desdemona and Cassio), one item (the handkerchief) and four locations (garden, bedroom, residence and palace). The initial state gives describes the relations between the characters (such as that Othello and Desdemona are married), their locations, their psychological attributes (such as that Iago is evil or that Othello is suspicious) and the attributes of the objects (the handkerchief is precious). The goal is that Othello kills Desdemona and, subsequently, himself.

Actions include physical actions (moving from one location to another, or taking, giving or dropping an item), the kill action and the communication actions. It is required that the character has a motive for performing an action. For example, the preconditions of $\mathsf{kill}(a, c, location)$ state that the following need to be true: $\mathsf{alive}_a, \mathsf{alive}_c, \mathsf{at}_{a,location}, \mathsf{at}_{c,location}, \mathsf{motive\text{-}for\text{-}dead}_{a,c}$. Similarly, for an a to give something an item i to c, a needs to wish c to posses i. In addition, there are *belief-revision* actions adopt-belief-loves and fall-in-love.

The axioms are used to reason about the state of the world (can-see and alone-at) and to determine motivations and mental states of characters. For example:

$$\begin{aligned}
\mathsf{motive\text{-}has}_{c,c,i} &\leftarrow \mathsf{greedy}_c = true, \mathsf{precious}_i = true, \mathsf{sees}_{c,i} = true \\
\mathsf{motive\text{-}has}_{c,c,i} &\leftarrow \mathsf{motive\text{-}has}_{c,a,i} = true
\end{aligned}$$

allows inferring that character c can be motivated to take action to achieve $\mathsf{has}_{c,i}$ if c is greedy and i is a precious item that c has layed eyes on, or if c desires some other character a to posses i. See other motivations (the axioms for motive-at and motive-for-dead) and characters' reasoning about other characters (the axioms for reason-to-believe-loves) in the Appendix for additional examples.

5.5.2 Kominis and Geffner

Kominis and Geffner [95] discuss planning in a multi-agent setting with partial observability (see Section 2.1) and *nested beliefs* – i.e. settings where agents are required to reason about beliefs of other agents.

One example domain is the Muddy Children problem [46]. In the problem n children are playing together. Each of the children wants to remain clean, but each would love to see others get dirty. Some of them (we denote this number by k), however, do get muddy. Each can see mud on the faces of others, but not on his own face. Their father comes along ans says "At least of you has mud on your forehead." The father then asks the following question over and over: "Does any of you know whether you have mud on your forehead?" It can be proven [46] that first $k - 1$ times he asks the question, all of the children will reply "No.", but when the question is asked k-th time, the children whose foreheads are muddy will reply affirmatively (the clean children will, at this point, still won't know whether they are muddy or not). Kominis and Geffener reformulate the problem by allowing one of the children (Active Muddy Child) to ask other children whether they know if they are clean or not (with all the other children listening to the response). The goal is for the Active Muddy Child to figure out whether they are muddy or not.

The author's approach works in two stages – they first formulate what they call *linear multiagent planning* problem and then they compile it into a classical planning problem with axioms. To formulate the linear multiagent planning problem, the authors use techniques developed for conformant and contingent planning problems [112][6] in single agent setting. This consists of search in a belief space, where beliefs are sets of states that the agent considers possible. This is combined these techniques with the approaches for representing beliefs in multiagent dynamic systems, namely Kripke structures [45] and dynamic epistemic logics [138]. This approach makes distinction between *objective formulas*, that are true in the world, and *epistemic formulas*, which define mental states of other agents. Multi-agent Kripke structures are triplets defined by a set of worlds, accessibility relations among the worlds for each agent and truth values that define the propositions that are true in each world. Accessibility relations define the truth conditions for epistemic formulas.

The actions that any of the agents can take falls into one these three categories:

- *Physical actions* – change the values of objective literals.
- *Sensing actions* – provides information about the objective literals to an agent. The information provided by the sensing action is private.
- *Communication* – provides the agent information about epistemic literals.

[6]See Section 2.1 for explanation of contingency planning.

Using the techniques that were developed for compiling conformant and contingent planning problems in a single agent setting [19, 112], the linear multiagent planning problem can be compiled to a classical planning problem. In the translation, epistemic literals become derived variables and their values are computed using axioms.

5.6 Computing heuristics

In deriving heuristics, we follow exactly the same approach as we did with numeric constraints and that can be summarised using the flowchart in Figure 1.1.

The heuristics that we are using are:

1. Heuristics based on the monotonic relaxation: h_{max}, h^+ and the disjoint landmark heuristic.
2. Abstraction-based heuristic or pattern database heuristics (PDBs).

For an overview of monotonic relaxation and abstraction in classical planning, see Sections 2.2.1 and 2.2.3, respectively. In both cases, we have *relaxed states* or *abstract states*, which can be viewed as representing sets of assignments of primary variables in the non-relaxed problem. As before, the key issue is how to evaluate conditions that involve secondary variables in a relaxed or abstract states. We apply three different approaches: the naive relaxation, evaluation using answer set programming, and evaluation using three-value semantics.

5.6.1 Naive relaxation

A very simple approach is what we call *naive relaxation*. Each axiom is treated as a zero-cost action, with the effect of the action being the head of the axiom and the precondition being the body. Initially, all derived variables are assigned their default value, *false*. Using this relaxation, any admissible classical planning relaxation computed on a relaxed problem gives us an admissible estimate for the problem with axioms.

Because the naive relaxation does not force the axioms to be applied, it is blind to the difficulty of achieving the negation of a derived proposition. The Fast Downward planner by Helmert [77] addresses this problem by using the following mechanism – for each derived variable y that occurs negatively (i.e. with the assignment of *false*) in some condition (goal, body of some axiom, or a precondition of some action), a new variable $\bar{y}$ and a set of axioms defining

$\bar{y}$ are added to the problem. If $A_y = \{y \leftarrow \varphi_i = \mathit{true} \mid i = 1, \ldots, k\}$ is the set of axioms that define y, we add an axiom $A_{\bar{y}} = \{\bar{y} \leftarrow \{\varphi_i = \mathit{false} \mid i = 1, \ldots, k\}\}$. As the default values for both y and $\bar{y}$ is *false*, admissibility is preserved. However, as the planner rewrites all the bodies of the axioms to disjunctive normal form (that is, $A_{\bar{y}}$ is rewritten as a set of axioms with $\bar{y}$ in the head of each of the axioms and different partial variable assignment in the bodies), this transformation often results in an exponential blow-up of problem size.

In many of the domains that we presented earlier, the naive relaxation is not informative. Consider the Min-Cut problem in Figure 5.2 – in any state of the relaxed problem, the goal is achievable at no cost by simply choosing *not to derive* $\mathsf{reachable}_t$ and applying the axiom $\mathsf{isolated}_t \leftarrow \{\mathsf{reachable}_t = \mathit{false}\}$.

5.6.2 Axiom-aware relaxations

To develop stronger relaxations, we take the same approach as we did in planning with switched constraints – we treat this as a question of consistency, which is delegated to an appropriate external solver. In case of planning with axioms, this is an *answer set programming* (ASP) solver.

Answer sets

Answer sets, also known as *stable models* provide declarative semantics for programs with negation as failure and epistemic disjunction [61]. (Answer set programs may use both negation as failure and classical logical negation. However, as we only use the former, we omit the later from the discussion.)

Definition 31. *A (ground)* answer set program *is a set of rules of the form*

$$l_0 \text{ or } \ldots \text{ or } l_k \leftarrow l_{k+1}, \ldots, l_m, \ \mathit{not}\ l_{m+1}, \ldots, \ \mathit{not}\ l_n$$

The left hand side is known as the head *and is a disjunction of propositions. The right hand side, known as the* body *is a conjunction of literals. The connectives not and or called* default negation *and* epistemic disjunction*, respectively.*

The intuitive meaning of this is that when all the literals in the body hold, then so does at least one of the propositions in the head.

A rule with an empty body is called a *fact*, and it asserts that at least one of the propositions in the head holds. A rule with an empty head is called a *constraint*, and it asserts that the literals in the body cannot all hold.

A *model* of a logic program is a set of true propositions that satisfies all of the rules of the program (propositions not in this set are false). A *stable* *model* or *answer set* is a model of a logic program that is minimal with the respect to set inclusion – i.e. it respects the negation by failure.[7] Hence, both negation in the body and disjunction in the head are interpreted as a form of defaults: p is false and $\neg p$ is therefore true, unless p is implied by a set of rules. A disjunction p *or* q is not exclusive, but as long as there is no other justification to infer that both p and q are true, the epistemic disjunction implies that at most one of them is true.

Given a planning problem with axioms, a state s can be represented as an answer set program $\pi(s)$. Axioms are rules of the ASP, while the assignment of primary variables variables is represented using facts: $(x = s[x]) \leftarrow p_\emptyset$. Because $\pi(s)$ is stratified, it is consistent and has a unique answer set [1]. The planning formalism defines the values of secondary variables in s as their values in the model. Hence, a secondary variable y is has the value *true* in s if $\pi(s) \cup \{\emptyset \leftarrow y = \mathit{false}\}$ is consistent.

The solver that we use to find stable models to answer set programs is Clasp by Gebser et al. [54].

5.6.3 Consistency-based monotonic relaxation

As we did for the switched constraints in Section 3.5.1, we extend MFDR (see Section 2.2.1) to accommodate for axioms. As already discussed, in a MFDR primary variables accumulate, rather than change values as actions are applied. In a relaxed state s^+ each primary variable has a set of values associated with it, $s^+[x_i] \subseteq \mathcal{D}(x_i)$, and s^+ itself is a set of states that can be obtained by assigning each variable x_i one value from its set $s^+[x_i]$. As before, action whose effect assigns value of a variable x_i to v in a relaxed application, adds v to the set of values associated with x_i.

We now explain how a relaxed state s^+ can be represented as an ASP. For each primary variable x we have a disjunctive fact,

$$(\bigvee_{v \in s^+[x]} x = v) \leftarrow p_\emptyset$$

and a set of *mutual exclusion constraints*,

$$\emptyset \leftarrow x = v, x = v' \text{for all} v, v' \in s^+[x] \text{such that} v \neq v'$$

to ensure that the variable has only one value. We denote the answer set program representing the relaxed state s^+ by $\pi(s^+)$.

[7]This is because the smallest set will only contain those propositions that have to be true according to the rules of the program. If there are no reasons to assign *true* to a proposition, it remains *false* and is left out of the set (i.e. as we *failed* to find a reason to assign *true* to the proposition, it is *false* by default).

Proposition 10. *Let s^+ be a relaxed state. $\pi(s^+) \cup \{\emptyset \leftarrow y = \mathit{false}\}$ is consistent if and only if y is true in some state in* states(s^+).

We will demonstrate this on Min-Cut example given in Figure 5.2:

Example 32. With actions $\mathsf{move}_{\mathbf{A},e_{15},e_{12}}$ and $\mathsf{move}_{\mathbf{A},e_{15},e_{56}}$, we reach a relaxed state such that $s^+[\mathsf{at}_\mathbf{A}] = \{e_{12}, e_{15}, e_{56}\}$ and $s^+[\mathsf{at}_\mathbf{B}] = \{e_{36}\}$. The program $\pi(s^+)$ has three stable models (one for each value of $\mathsf{at}_\mathbf{A}$) and $\mathsf{reachable}_6$ is true in all, since no matter where roadblock **A** is placed the target node 6 is reachable as long as **B** does not move. Hence, $\pi(s^+) \cup \{\emptyset \leftarrow \mathsf{isolated}_6 = \mathit{false}\}$ is not consistent.

We may add to $\pi(s^+)$ some further constraints, such as mutual exclusion between primary variable assignments. This excludes from states(s^+) certain states that are not reachable in the original planning problem, thus strengthening the relaxation. It is the same idea as *constrained abstraction*, used to improve pattern database heuristics, as described by Haslum et al. [70].

As shown on the diagram in Figure 1.1, the monotonic relaxation is used to build a relaxed planning graph and compute the admissible heuristics. As with planning with switched constraints, we compute h_{max}, h^+ and the disjoint landmark heuristic. We use exactly the same procedures to compute those heuristics as we did for numeric constraints (see Sections 3.5.3, 3.5.4 and 3.5.5, respectively).

5.6.4 Consistency-based abstraction

Again, we employ projection (see Sections 2.2.3 and 3.5.2) to build PDBs. Our approach is the same as with switched constraints – abstract states can be thought of as relaxed states in which the variables in the pattern have only a single value, while variables not in the pattern have all of the variables in their domain. Therefore, the abstract state s^A can be formulated as an answer set program, $\pi(s^A)$ and the consistency can be checked in the same way as before (that is, using an ASP solver). As an illustration, we use the problem from the Min-Cut domain.

Example 33. Consider again the Min-Cut problem in Figure 5.2, and a projection on the single-variable set $A = \{\mathsf{at}_\mathbf{A}\}$. The abstract state has $s^A[\mathsf{at}_\mathbf{A}] = \{e_{15}\}$ and $s^A[\mathsf{at}_\mathbf{B}] = \{e_{12}, e_{14}, e_{15}, e_{23}, e_{26}, e_{36}, e_{45}, e_{56}\}$, i.e., the set of all edges. Again, the program $\pi(s^A) \cup \{\emptyset \leftarrow \mathsf{isolated}_6 = \mathit{false}\}$ is not consistent since node 6 remains reachable no matter where **B** is as long as **A** stays on edge e_{15}.

As stated in Section 3.5.2, PDBs are computed by exhaustive reverse exploration from the abstract goal states. This approach is not easily adapted to problems with axioms for the same reason it is not easily adapted for switched constraints – goal is a partial variable assignment over both primary and secondary variables, so it is not straightforward to find all the goal states. Reverse exploration is also difficult as action preconditions are partial variable assignments over both primary and secondary variables, so finding the states in which an action can be applied is more difficult than in classical planning without state constraints.

As with switched constraints, we use a two-stage PDB computation already described in Section 3.5.2. Hence, building an axiom aware PDB takes more time than the standard PDB construction. An advantage of PDB heuristics in this setting is, however, that this overhead is limited to the precomputation phase only; state evaluation is done by a table lookup, and takes no more time than in a standard PDB heuristic.

5.6.5 Exploring weaker relaxations

Invoking an ASP solver each time we need to test consistency of a given condition is time consuming. Although the heuristics we obtain are more informed than the ones we will describe in this section, the time needed to compute the heuristics means that, in most cases, they are not effective (as we will see in the experiments). For this reason, we also developed a weaker relaxation. As long as consistency test is *sound* and we treat the secondary condition as unsatisfied only when it is *proven* inconsistent with the relaxed state, our relaxation, and the heuristics we derive from it, are still admissible.

In this relaxation, we use three value semantics to determine the truth of a given condition. Given a relaxed state s^+, we treat a value assignment $s^+[x] = v$ in a relaxed state as

- *false* if $v \notin s^+[x]$,
- *true* if v is the only value in the set, $s^+[x] = \{v\}$ and
- *unknown* otherwise.

We evaluate compound formulas using the rules given in Figure 5.5 (partial variable assignments are evaluated as a conjunction of single variable value assignments). We then apply the stratified fixpoint evaluation procedure (given in Figure 5.1) to compute a value in $\{\mathit{false}, \mathit{true}, \mathit{unknown}\}$ for each secondary variable. A secondary literal y is considered true if it evaluates to *true* or *unknown*. This is sound, in that y remains *false* only if

$A \wedge B$	T	U	F
T	T	U	F
U	U	U	F
F	F	F	F

$A \vee B$	T	U	F
T	T	T	T
U	T	U	U
F	T	U	F

$\neg A$	T	U	F
	F	U	T

Figure 5.5: Three value semantics.

y is false under every interpretation of the unknown propositions, and thus only if $\pi(s^+) \cup \{\emptyset \leftarrow y = \mathit{false}\}$ is inconsistent. While this evaluation is sound, it is not *complete* (i.e. guaranteed to derive the condition as false iff it is false in every state belonging to states(s^+), as stated in Proposition 10), making the relaxation weaker than the ASP-based relaxation. Again, we can demonstrate this on an example:

Example 34. Consider again the Min-Cut problem in Figure 5.2, and the relaxed state $s^+[\mathsf{at}_\mathbf{A}] = \{e_{12}, e_{15}, e_{56}\}$ and $s^+[\mathsf{at}_\mathbf{B}] = \{e_{36}\}$ from Example 32. Since $\mathsf{at}_\mathbf{A} = e_{12}$ and $\mathsf{at}_\mathbf{A} = e_{56}$ both evaluate to *unknown*, we can derive that $\mathsf{isolated}_6$ is also *unknown*. Thus, the relaxation fails to prove that moving roadblock **A** alone is insufficient to reach the goal, unlike the stronger relaxations shown in Examples 32 and 33.

This is the same as the relationship between the strong relaxation in planning with numeric constraints and its two weaker approximations, presented in Section 3.5.1. A partitioned condition φ holds in the strong relaxation iff there is a state in states(s^+) in which φ holds. In contrast, φ might hold in s^+ under intermediate (or weak) relaxation, even if there is no such state. However, if φ does not hold under intermediate (or weak) relaxation, this implies that there is no such state in states(s^+). Similar idea can also be found in the work of Francès and Geffener [51], who also develop weaker, but computationally tractable relaxation using incomplete local consistency algorithms, as an approximation to their original relaxation (see Section 2.3.2).

5.6.6 Extending the relaxations to discrete finite domain secondary variables

The ASP-based relaxation and the relaxation based on three value semantics can be extended to discrete finite domain secondary variables in the following ways:

1. For the ASP-based relaxation, we make two changes to the program. First, for each secondary variable, we need to add a disjunctive fact,

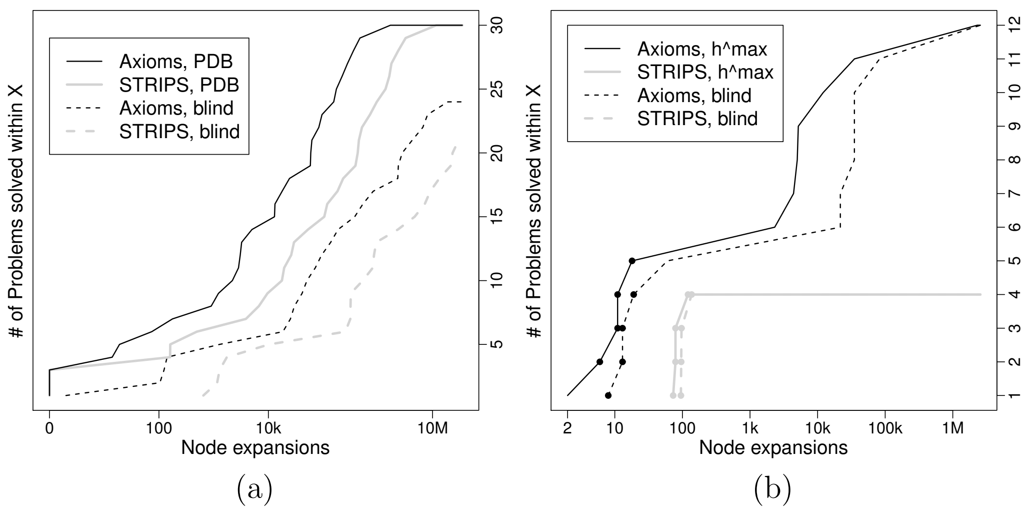


Figure 5.6: Node expansions required to prove optimality on equivalent problem formulations with and without axioms: (a) Sokoban problems; (b) controller verification problems due to Ghosh et al. [2015]. The door controller problems are marked with a dot (●) in (b).

$(\bigvee_{v \in s^+[x_s]} x_s = v) \leftarrow p_\emptyset$, and a set of mutual exclusion constraints same as for the primary variables. Second, Proposition 10 changes in the following way – given a relaxed state s^+ and a variable y with the domain $\mathcal{D}(y) = \{x_1, \ldots, x_n\}$, the program $\pi(s^+) \bigcup_{j=1,\ldots,n, j \neq i} \{\emptyset \leftarrow y = x_j\}$ is consistent if and only if y is x_i in some state in states(s^+).

2. The three value semantics based relaxation can be extended by assigning each secondary variable a set of values. If the body of the axiom evaluates to *true*, we assign the variable the value in the head of the axiom. If it evaluates to *false*, we assign the variable its default value. If the body evaluates to *unknown*, we assign it the set consisting of its default value and the value given in the head of the axiom.

5.7 Experiments

Here we will present results of the experiments. We will show that reformulating the problems using axioms eliminates the unnecessary choices from the solver, therefore leading to smaller state space and shorter plans, using the Sokoban and the controller verification domains. We will also compare the above described heuristics with each other and with blind search in terms of nodes expanded and total time to solve each of the problems.

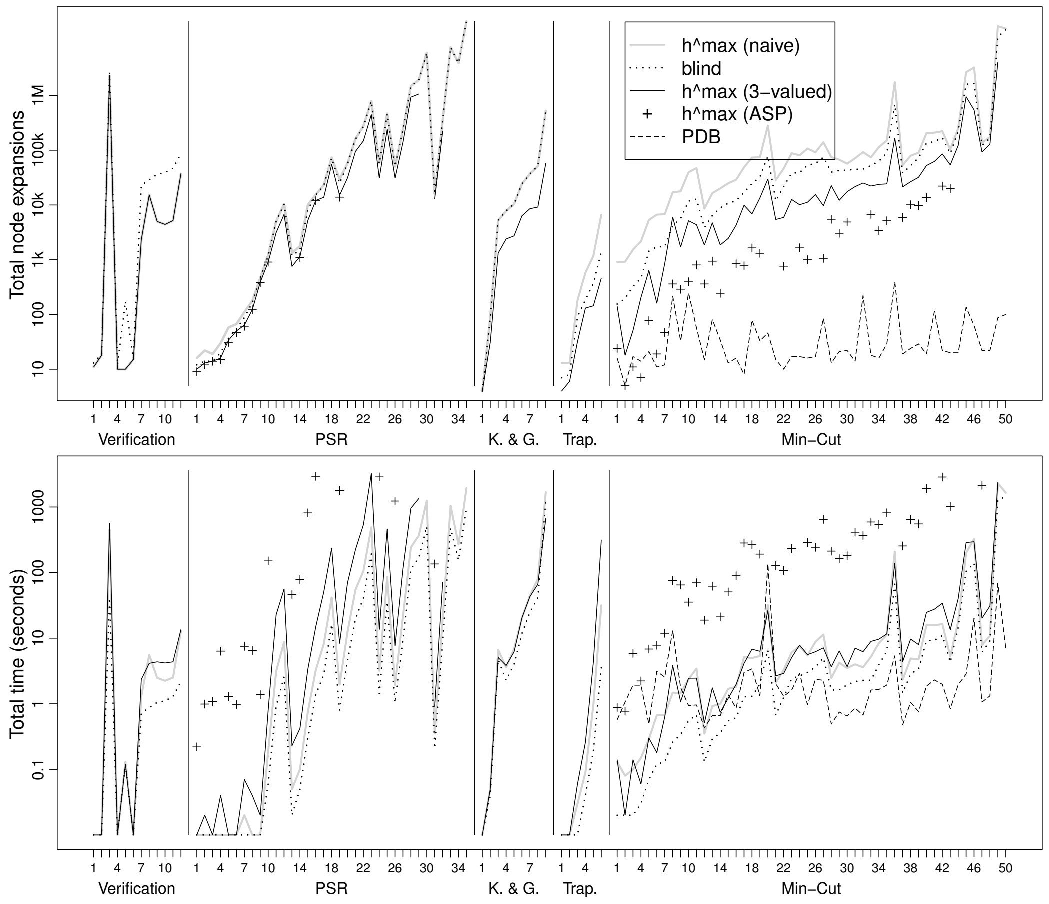


Figure 5.7: Total nodes expanded (above) and total planning time (below) with different heuristics. Instances in each set are sorted by increasing shortest plan length.

5.7.1 Sokoban

Figure 5.6 shows the effect that reformulating the Sokoban problem using axioms (as opposed to having explicit "move" actions) has on A* search. The graph shows the number of node expansions needed to prove the optimality, i.e. to reach the $f^\star$-value[8]. (This is a function of size of the search space and informedness of the heuristic only, not subject to the tie-breaking variations.) In our experiments, we compared the STRIPS formulation used in IPC 2008 (that is, without axioms) with our formulation that uses axioms. The pattern database heuristic used is the canonical additive combination of several PDBs [72]. For the formulation with axioms, we use one PDB per stone location variable. In the STRIPS formulation, the location and "at-goal" status of each stone is split over variables, so we included both in each PDB to get an equivalent heuristic. CPU time was limited to 1 hour and memory to 3GB per problem.

Blind search on the STRIPS formulation expands, on average, 17 times more nodes than the formulation with axioms. However, node expansion with axioms is, on average, an order of magnitude slower, so runtimes end up within a factor of 2.7 of each other. The precomputation time for the axiom aware PDB heuristic is also several orders of magnitude larger.

5.7.2 Controller verification

The advantage of using axioms can also be seen in the controller verification domain – again, unnecessary actions and choices are removed, resulting in smaller state space. Figure 5.6 shows the comparison between the compilation with axioms and the STRIPS compilation on two sets of problems from Ghosh et al. [66], testing several safety properties of a door lock and an ACC systems. Some of these problems have no plan – we consider these problems as "solved" when the planner proves the non-existence of a plan.

The door lock system is relatively small, with only 6 control actions and 8 environment actions. The ACC example is much bigger, with 34 control actions and 691 environment actions. Neither heuristic nor blind search were able to solve any of the ACC problems if given STRIPS formulation, even with 4 hours CPU time and 60GB of memory. Using formulation with axioms, in contrast, solves all of the problems using blind search, with even the most difficult ones taking less than a minute.

[8] $f^\star$ is the optimal cost. Terminology related to A* will be given in Section 6.1.

5.7.3 Experiments with weaker relaxations

The impact of different relaxations on the search is shown in Figure 5.7. We used 5 sets of problems:

- The verification problems from Ghosh et al. (Section 5.4.1) [66]
- The PSR domain (middle-size set from IPC 2004)
- Multi-agent planning problems from Kominis and Geffner [95] (Section 5.5)
- Instances of the trapping game (blocker) (Section 5.4.2), played on graphs with 4 to 47 nodes and
- Random instances of Min-Cut domain, with graphs of size 12–20 and 3–4 roadblocks (Section 5.3.1).

Each planner was run with up to 1h CPU time and 3GB memory per problem. Our implementation is built on Fast Downward planner [78] and uses Clasp v2.1.3 as an ASP solver.

If there are no assignments of *false* to secondary variables in the bodies of the axioms, the naive version of $h^{\max}$ coincides with 3-value semantics $h^{\max}$ (here, this happens only in the verification problems). For all the other problems, the naive relaxation is as uninformed as blind search. The ASP-based relaxation is more accurate than the 3-value semantics relaxation only when some secondary facts can be derived from the disjunction in a relaxed state. In our test set, this occurs only in the PSR and Min-Cut domains. However, the weaker relaxation is much faster to compute. (Results for the ASP-based $h^{\max}$ heuristic are omitted from the other domains, in which it expands exactly the same number of nodes but takes far more time to do so.) Although all the heuristics except the naive reduce the amount of search, blind search is still often the fastest; its main limitation is memory. This has also been demonstrated in STRIPS planning, e.g. in results from the IPCs [9]. The 3-valued $h^{\max}$ heuristic is faster than blind search on the hardest of Kominis and Geffener's problems and results with axiom-aware PDB heuristic on the Min-Cut domain show that even the ASP-based relaxation can be sufficiently informative to compensate for the overhead of computing it. We did not try the PDB heuristic on other problems since we do not know which are good abstractions, if indeed any exist.

[9]`http://icaps-conference.org/ipc2008/deterministic/`

Chapter 6

Preferred operators in partial expansion A*

To compute optimal plans, we use A* search guided by an admissible heuristic. We have, however, also developed a new search algorithm which can result in significant savings in terms of memory and time when the heuristic is expensive to compute, yet accurate and state have many successors. This situation is characteristic of many planning problems, including the PSR problem (Section 3.3.2). In the PSR problem, each state has at most as many possible successors as there are switches in the network – the number of switches can be in the order of thousands (e.g. transmission network for New South Wales), and in the network used in our experiments there were 45. While not all of those successor states are necessarily valid, they still need to be generated before we can determine that they are invalid. The algorithm that will be presented in this chapter can save us from generating some of those states as well.

Our algorithm, which we call PrefPEA*, combines *preferred operators* with *partial expansion A**. While preferred operators have been used in non-optimal planning, they have not been combined with optimal search algorithms such as A*. In our case, the preferred operators are obtained as a side effect of computing h^+. While h^+ is expensive to compute, the optimal plan for a relaxed problem can provide us with more information than just heuristic cost estimate. We only partially expand the nodes, therefore and avoid generating all of the successors and reduce the number of heuristic computations (as well as computation necessary for determining whether the states are valid).

Techniques that we build on will be described in Section 6.1. In Section 6.2 we will explain how we combined those two ideas to reduce the number of heuristic evaluations in cases where the heuristic is informative yet expensive to compute. Section 6.3 will present the experiments and results.

6.1 Background

Here, we will give an overview of search techniques that we build on, namely identifying *preferred* or *useful actions*, and *partial expansion A** (PEA*).

For the description of the A* see the original paper by Hart et al. [111]. We assume that the reader is familiar with the algorithm and the terms used. Here, we will just give a few definitions of the terms used in this chapter, all which are borrowed from Hart et al.'s paper. For a given node n, cheapest known cost of reaching n from the start state is denoted $g(n)$. *Heuristic cost estimate* of n is denoted $h(n)$ and it estimates the minimum cost of reaching the cheapest goal state from the given state. If the heuristic is known to never overestimate the cost, it is an *admissible heuristic*. As stated earlier, admissible heuristics are used in optimal planning because certain optimal search algorithms, like A*, guarantee that the solution returned is optimal, provided that the heuristic is admissible. *Evaluation function*, or $f(n)$, is the sum $g(n)$ and $h(n)$ values. The cost of the cheapest path to a goal state is denoted $f^\star$. A* can be described as best-first search on the f-value.

A* is a graph search algorithm, but in this work we are interested in presenting and solving the planning problems as state-space search. Therefore, our graph is a state-space of the planning problem, and our successor operators are actions. We will use the term *node* to refer to a tuple $n = \langle s, g(s), h(s)\rangle$. We define $g(n) = g(s)$, $h(n) = h(s)$ and $f(n) = f(s)$.

6.1.1 Preferred actions in non-optimal search

Identifying preferred actions has been developed and used with non-optimal search techniques such as hill-climbing and greedy search (see Richter and Helmert [120] and Hoffmann [81]). The preferred actions in state s are actions in a relaxed plan computed from s that are also applicable in s. The intuition behind this idea is that, if the relaxed plan is similar to the actual plan, then choosing an action that is also part of it is more likely to be a step towards the goal and giving preference to successor states generated from those actions can lead to the goal state more quickly.

6.1.2 A* with partial expansion

An optimal search algorithm such as A* must expand any state that could possibly lie on a cheaper path to the goal. That is, any state whose f-value is less than the optimal f-value must be expanded. In problems with large branching factor, this leads to heavy memory requirements. To tackle this challenge, Yoshizumi et al. [143] developed PEA* algorithm, originally applied for multiple sequence alignment problem in genome informatics. Their algorithm tries to avoid placing unpromising states on the open list, by expanding states only partially and re-inserting them on the open list for later consideration with lowered priority. This reduces the memory requirements and is useful in cases where the branching factor is large.

The algorithm works as follows – they introduce a (predefined and non-negative) cut-off value parameter C (which is constant) and an additional value F associated with every node (which is separate from the f-value in simplest form of A*). Initially, when a node n_0 is selected for expansion, the F-value of n_0 set to its f-value. After the node is expanded, a child node n_1 is stored in the open list only if $f(n_1) \leq F(n_0) + C$. In that case n_1 is considered a *promising child*. Otherwise, n_1 is considered unpromising, and is not stored. If n_0 has any unpromising children, the F-value of n_0 is then set to the smallest f-value among all the unpromising children of n_0 and n_0 is stored in the open list with the new F-value recorded (if there are no unpromising children, n_0 is placed in the closed list). While A* expands nodes on the open list in incremental order of f-value, here nodes are expanded in the incremental order of F-value. If n_0 has no promising child nodes, then it does not store child nodes at all and only revises the parent's F-value to the lowest f-value among unpromising child nodes (this lowers the priority of the parent node for expansion).

When C is infinity, this algorithm becomes A*. With $C = 0$, it never stores nodes whose f-values are greater than the optimal cost and the memory requirement becomes the same as the closed list of A*. This makes their algorithm very effective when the ratio of open to closed list is large. Yoshizumi et al. give the following example – suppose that the state space forms a tree with the branching factor of b. The ratio of open to closed list size becomes $b - 1$ to 1 and consequently, with $C = 0$, the memory requirements are reduced by a factor of b (as the algorithm only needs the same memory as the closed list). In their multiple sequence alignment problem experiments, they report reducing the memory requirement by a factor of few hundred (in best case).

6.1.3 PEA* with selective node generation

This form of PEA*, however, still generates and evaluates all successor states of an expanded state to determine which are promising. Despite the memory benefits of using the basic form of PEA*, it incurs a time overhead, as it repeatedly generates all the children of a given node each time the node is expanded. Thus, while memory is always saved, time trade-off exists. Felner et al. [47] further improved the algorithm by using a problem- and heuristic-specific procedures to determine the successor set without generating and evaluating all the successors. They called their algorithm *Enhanced PEA** (EPEA*). The idea is to, when expanding a node n, only generate children whose f-value is the same as $f(n)$.

This works in domains in which the operators applicable to n can be classified by the change in f-value (Δf) of the children of n they generate. EPEA* requires defining a domain-dependent *Operator Selection Function* (OSF), which gets a state p and a value v as an input and returns a list of operators that, when applied to p, lead to a state whose difference in f-value from p is v. Additionally, it returns v_{next}, that is the next Δf in the set of operators applicable to p.

When a node n is expanded with stored value $F(n)$, $F(n)$ might be larger than $f(n)$, in cases where n has already been expanded in the past and it has inherited $F(n)$ from one of it's children. We only want to generate a children with $f(c) = F(n)$, so we need operators with $\Delta f = F(n) - f(n)$. Operator selection function is then called with node n and Δf to identify such operators. Children with the f-value of $f(c)$ are generated, and n is returned into the open list with the redefined value of $F(n) = f(n) + v_{next}$. If no larger value is possible (so all children nodes have been generated), the node is placed on the closed list. If the goal is found before n is placed on the closed list, EPEA* never generates any nodes with suboptimal f-value.

6.2 Algorithm

We adopted the idea of partial expansion, but staged the node expansion by preferredness of successors instead of by f-value. Pseudo-code is shown in Figure 6.1. When a heuristic value is calculated, as a side-effect, a set of preferred actions associated with that node is found and stored with the state. I.e. a node becomes a tuple $\langle s, g(s), h(s), \mathsf{pref}(s) \rangle$, where s is a state, $g(s)$ is the g-value of s, $h(s)$ is the h-value and $\mathsf{pref}(s)$ is the set of preferred actions. In our experiments, the heuristic is h^+ and the set of preferred actions are the actions comprising the optimal delete-relaxed plan.[1] When a node is selected for expansion, we generate and evaluate only one preferred successor (Line 8), using one of the actions in the preferred set. The action is then removed from the set (Line 7) of preferred actions associated with the

[1]It should be noted that this algorithm can work with other heuristics, as long as the heuristic identifies a set of preferred successor operators.

parent node and the parent node is kept on the open list. If the node that is selected for the expansion has no preffered operators left, then the then we generate all the remaining successor states by using the set of non-preferred operators (Lines 10–13). The node is then moved to the closed list.

The rules for choosing a node to expand are shown in Line 4. We prioritise the nodes that still have unexplored preferred successors. This is done using non-emptiness of the preferred action set as an additional tie-breaking criterion, after the standard tie-breaking in favour of lower h-value. That is, if two nodes have equal f-values and equal h-values, but one of them has a non-empty set of preferred actions, that node is chosen for expansion.

6.3 Experiments

The benefit of our algorithm is limited to avoiding heuristic evaluations in the same layer as the optimal f-value (see Figure 6.2). This is most useful when states have many successors and the heuristic is accurate. In cases where, on the other hand, the heuristic is inaccurate or the branching factor is small, the savings are small as well. The experiments on the HBW domain and the PSR domain illustrate both cases (the domains were described in Sections 3.3.1 and 3.3.2, respectively). While there are substantial savings in the PSR domain, the savings for hydraulic blocks world, where the branching factor is lower, are much lower. Because PrefPEA* sometimes expands nodes in different order than A*, and because the remaining ties are still broken arbitrarily, it is possible for PrefPEA* to be "unlucky" and expand more nodes than A*, even if the aggregate results are better. This was observed in small number of cases (less than 2%).

For the PSR domain, we use the semi-rural network from Thiebaux et al. [134] and generate 171 problems with one, two or three faults. The network consisted of 7 generators, 45 buses and 26 switches. The faults were generated by chosing the buses at random. We used constant (state-independent) cost of one for each action. For the hydraulic blocks world domain, we used 80 problems with between 4 and 7 blocks and 3 to 5 cylinders. Parameters such as block weights, cylinder heights and areas, etc. were set randomly, with the aim of creating problems that are solvable, but with the state constraints forcing the plans to be different than in the unconstrained case. The addition of the limit on the number of towers and the state constraints makes this problem much harder than the usual STRIPS Blocksworld. (Blind A* solves only 4 out of 20 6-block problems, while A* search with h^+ heuristic solves 17.)

With a 30 minute timeout for each problem and a plain A*, the planner

```
1: procedure PrefPEA$^\star$
2:   Set $open = \{\langle s_0, 0, h(s_0), \mathsf{pref}(s_0)\rangle\}$, $closed = \emptyset$.
3:   while $open \neq \emptyset$ do
4:     Select $n = \min_{\prec} open$, where
       $(n \prec n') \equiv \quad (f(n) < f(n'))$
                      $\vee\ (f(n) = f(n') \wedge h(n) < h(n'))$
                      $\vee\ (f(n) = f(n') \wedge h(n) = h(n')$
                         $\wedge\, \mathsf{pref}(n) \neq \emptyset \wedge \mathsf{pref}(n') = \emptyset))$
5:     if $n$ is a goal state then return $n$.
6:     if $\mathsf{pref}(n) \neq \emptyset$ then
7:       Select $a \in \mathsf{pref}(n)$, remove $a$ from $\mathsf{pref}(n)$.
8:       Generate $s'$ from $n$ through $a$.
9:       NewState$(s', g(n) + \mathrm{cost}(a))$
10:    else
11:      for each non-preferred successor $(a', s')$ of $n$ do
12:        NewState$(s', g(n) + \mathrm{cost}(a'))$
13:      Move $n$ to $closed$.
14:  return $null$.

15: procedure NewState$(s, g)$
16:   if $\nexists n' \in open \cup closed$ with state $s$ then
17:     Add $\langle s, g, h(s), \mathsf{pref}(s)\rangle$ to $open$.
18:   else if $g < g(n')$ then
19:     Set $g(n') = g$ and update parent pointer.
20:     if $n' \in closed$ then Move $n'$ back to $open$.
```

Figure 6.1: Partial Expansion A* with Preferred Actions. The NewState subroutine handles updating of path cost and node re-opening, as in standard A*.

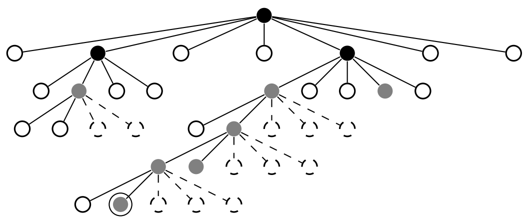


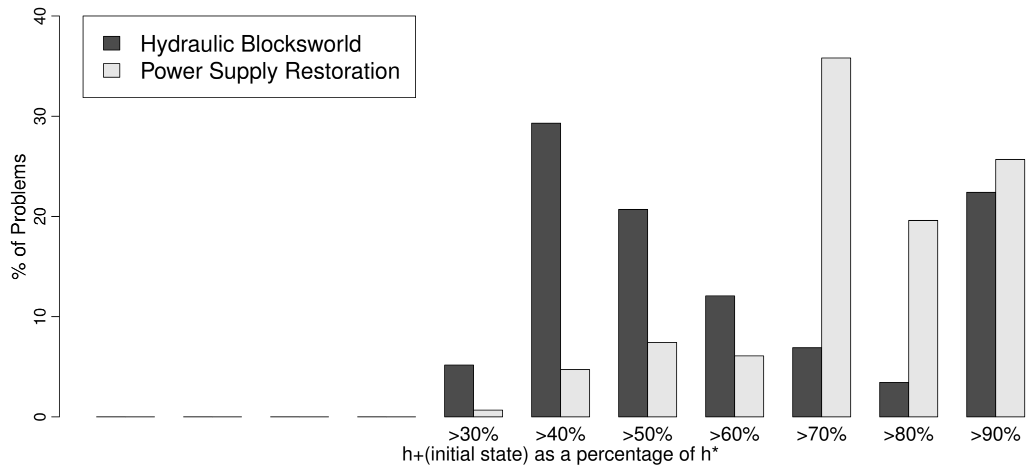
Figure 6.2: Illustration of PrefPEA*. Black nodes have $f(n) < f^\star$; these must be fully expanded. Gray nodes have $f(n) = f^\star$; some of these will be expanded, and may be partially expanded. White nodes have $f(n) > f^\star$. The dashed part represents non-preferred successor nodes that are never generated or evaluated. Once the search has reached the $f^\star$ layer and hit a node on an optimal path, tie-breaking on h will keep it on this path (assuming no zero-cost actions). From this point, only preferred successors are generated.

solver only 150 PSR problems and 62 HBW problems. Figure 6.3 shows how close the heuristic estimate of initial state, $h^+(s_0)$, is to the actual optimal f-value of the final state as a percentage. It was found to be more accurate in the PSR problems than in the HBW problems. Additionally, the average branching factor in PSR is 26.7, while in HBW it is 1.99. Consequently, the reduction in number of state evaluations is much higher in PSR problems than in HBW problem (see Figure 6.3). In PSR, using PrefPEA* reduces the number of evaluated states by 42.8% compared to A* (aggregated on the problems solved with both), and over 90% in over a quarter of the instances. Since computing the h^+ heuristic accounts for 95% of the total runtime (on average), in this domain, this translates to a roughly proportional 41.6% reduction in aggregated runtime. In HBW, the aggregate reduction in evaluations is only 3.4%, and, as heuristic evaluations are much faster in problems in this domain (only 16.2% of the total time on average), this does not lead to any reduction in the total runtime.

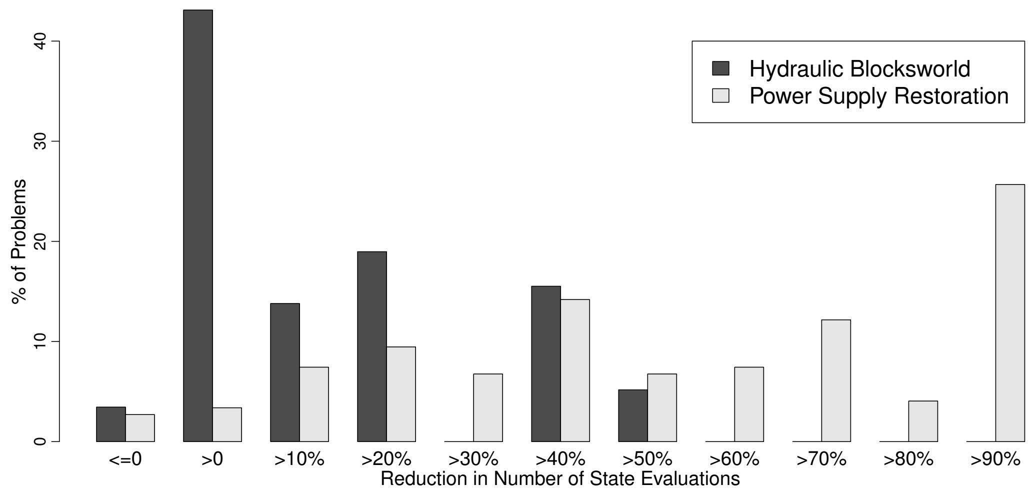
6.4 Related work

Given the high memory requirements of the original A*, we are not the first to attempt to create more memory-efficient, yet still optimal alternatives.

One example is the iterative deepening A* (IDA*) [96], which, unlike ordinary A*, does not maintain an open list of nodes, other than the ones on the current path (it works the same as iterative-deepening depth-first search [96], except it uses the f-value as a cutoff). It's memory requirement



(a)



(b)

Figure 6.3: (a) Accuracy of the relaxed plan heuristic, measured by $h^+(s_0)$ as a percentage of f^*. (b) Distribution of the reduction in number of state evaluations using PrefPEA* compared to plain A*. (A "reduction" < 0 means PrefPEA* evaluates more states, as a result of expanding more nodes. This increase is never more than 15%.)

is linear in the length of the solution. Another example is the recursive best-first search (RBFS) [97], which keeps track of f-value of the best alternative path available from any ancestor of the current node. If the node being evaluated exceeds this value, the search continues from the best alternative path. The memory requirement is linear in the size of the optimal solution. A shortcoming of IDA* and RBFS is that they might need to potentially re-explore same nodes many times (and therefore have greater time complexity than A*). Another weakness is that they do not utilise all of the memory available (which could be used to store more nodes and avoid as much re-exploration of same paths). Memory-bounded A* [23] and simplified memory-bounded A* (SMA*) [125] were, on the other hand, created to reduce the number of nodes re-generated by filling in all of the available memory. SMA* works just like ordinary A* except, after it fills in all of the memory, it drops the leaf node with the highest f-value and (like RBFS) backs up the value of the forgotten node to it's parent.

Unlike these alternatives, our algorithm never has to re-expand a node, as all of the visited nodes are kept in the memory. It therefore makes saving both in terms of memory and time requirements.

6.5 Future work

So far, we tested the procedure only on a small set of problems and only on the domains with numeric state constraints. The algorithm is, however, more general – while it works well on our PSR problems, there is no reason while it couldn't be employed on classical planning problems or other kind of search problems. To properly evaluate the algorithm, it should be tested on a larger set of domains and problems, for instance the problems from the International Planning Competition[2].

Besides h^+, it could be tested with other heuristics. For example, we could use abstraction heuristics. Given a state, there exists a set of actions in the abstract space that achieve the goal from the corresponding abstract state. These could be used as a set of preferred operators. However, if an operator in the abstract space corresponds to multiple operators in the non-abstract space, we would need to make the choice of which non-abstract operator to place in the set.

[2]`https://helios.hud.ac.uk/scommv/IPC-14/domains\_sequential.html`

Chapter 7

Conclusion

In this thesis we demonstrated how state constraints can be used to model a variety of problems. The concrete examples of state constraints that were introduced were numeric constraints and axioms. In the first case, we presented state constraints as a construct that allows us to combine methods from classical planning with solvers for other kinds of problems. An example are interconnected physical systems whose behaviour can be modelled by a set of equations. While efficient solvers for these constraints exist, they might not be able to effectively deal with the planning aspect of the problem. We presented how we can enable a planner to utilise those tools. In the second case, we formulated some existing domains from classical planning using axioms, which are a type of state constraints, making the domain descriptions more elegant and the problems easier to solve (which is a consequence of removing unnecessary choices from the planner).

We adapted the well-known techniques from optimal classical planning to this setting. We used the consistency-checking techniques to evaluate conditions involving secondary variables in a relaxed state. For both numeric constraints and axioms we developed relaxations of different strengths – the computationally expensive complete check, and incomplete, but sound and computationally tractable relaxations. This allowed to build a relaxed planning graph, which we used to compute constraint-aware versions of admissible heuristics h_{max}, h^+ and disjoint landmark heuristic. We demonstrated that an abstract state is equivalent to a relaxed state, so the same method allowed us to reason about abstract states. This, in turn, enabled us to compute pattern database heuristics.

We discussed compilation of problems with state constraints to formulations without them. We have shown that planning with switched numeric constraints can be reduced to classical planning, although potentially at exponential increase in problem size. The same observation has already been made by Thiébaux et al. [136] for axioms.

Motivated by the need to properly evaluate plans in power reconfiguration domain, we discussed the issues related to the action cost being a function of an extended state. In this domain, all secondary variables affecting the cost have a unique assignment of values in a given state. This simplifies the problem as, although the variables involved are still calculated using state constraints, we don't need to find the cost in the cheapest extended state. We modified h^+ heuristic for this problem, but our approach was unfortunately not efficient enough to be useful for search. Efficient computation of heuristics in this setting remains an open problem.

We modified the A* heuristic by combining the idea of preferred operators with partial expansion A*. The resulting algorithm, which we call PrefPEA*, works well in cases where the heuristic is informative and the states have many successors. It requires that the heuristic used also returns a set of preferred operators (in case of h^+, these are the actions that make up the optimal relaxed plan). In the PSR domain, PrefPEA* reduces the number of evaluated states by 42.8% compared to A* (aggregated on the problems solved with both), and over 90% in over a quarter of the instances.

7.1 Alternative frameworks and future work

The approach to planning with state constraints presented here was relatively narrowly focused – the only technique that we adapted was optimal planning using state-space search guided by an admissible heuristic. State-space search was used as it is the most commonly used way of optimally solving classical planning problems [79]. This leaves plenty of room for investigating possible alternatives. In Chapter 2, we already mentioned a paper by Scala et al. [126] which proposes solving problems by compilation to SMT. Other potential approaches might include, for example, plan space planning, compilation to optimisation problems such as MIP or use of constraint-satisfaction techniques. Focus on optimal planning was chosen because in many of the domains that we consider (such as PSR and controller verification), shorter plans are preferable. In many cases, however, end user might be satisfied with sub-optimal or near-optimal solutions. These can be generated if the heuristic is non-admissible (such as additive heuristic [16], FF-heuristic [85] or pairwise max heuristic [106]), or if the non-optimal search algorithm is used (examples include weighted $A^\star$ [116], explicit estimation search [133] other forms of bounded suboptimal search [75]).

Appendix A

Proofs

Proofs of Proposition 1 and Proposition 2 given in Section 3.4 are presented here. These proofs are to appear in our JAIR paper [73].

A.1 Proposition 1

Proposition 11. *Let $\langle \varphi_P, \varphi_S \rangle$ be a partitioned condition. There is a formula $\mathrm{F}(\varphi_S)$ over V_P such that for every state s, $s[\varphi_P \wedge \mathrm{F}(\varphi_S)] = true$ if and only if $\langle \varphi_P, \varphi_S \rangle$ holds in s.*

Proof. Let

$$\mathit{True}(\varphi_S) = \{ s \mid C_P(s) \cup \varphi_S \cup C_{\mathsf{inv}} \ \textit{is satisfiable} \ \}$$

$\langle \varphi_P, \varphi_S \rangle$ holds in state s if and only if $s[\varphi_P] = \mathit{true}$ and $s \in \mathit{True}(\varphi_S)$ (by Definition 18). Since states are assignments of values to the primary state variables, each of which has a finite domain of values (Definition 13), the set of possible states is finite (though exponentially large) and hence so is $\mathit{True}(\varphi_S)$. Thus we can write a formula, $\mathrm{F}(\varphi_S)$, over V_P, that is true exactly in the states $\mathit{True}(\varphi_S)$. This formula can be written simply as a disjunction of partial variable assignments, each defining a complete state in $\mathit{True}(\varphi_S)$, but more compact forms may also exist. Then $\varphi_P \wedge \mathrm{F}(\varphi_S)$ characterises exactly the states in which $\langle \varphi_P, \varphi_S \rangle$ holds. □

A.2 Proposition 2

Proposition 12. *Let φ be any formula over the primary variables V_P. There exists a set of switched constraints C such that for every state s, $\langle p_\emptyset, C \rangle$ holds in s if and only if $s[\varphi] = true$. In addition, the size of C is polynomial in the size of φ.*

Proof. Without loss of generality we can assume φ to be in negation normal form, since translation to this form does not increase the size of the formula more than polynomially.

We will introduce a secondary variable $0 \leq p_\psi \leq 1$ for every subformula ψ of φ, along with a set of constraints C' such that

$$\{p_\psi > 0\} \cup C' \cup C_P(s) \text{ is satisfiable iff } s[\psi] = \mathit{true} \qquad \text{(A)}$$

The constraint set C claimed by the proposition is then given by $C = \{p_\varphi > 0\} \cup C'$. The construction of C' is as follows:

- For each assignment of the form $v = e$, C' contains the two switched constraints $v = e \rightarrow p_{v=e} = 1$ and $v \neq e \rightarrow p_{v=e} = 0$.
- For each conjunctive subformula $\psi = \chi_1 \wedge \ldots \wedge \chi_k$, C' contains the constraints $p_\psi \leq p_{\chi_1}, \ldots, p_\psi \leq p_{\chi_k}$.
- For each disjunctive subformula $\psi = \chi_1 \vee \ldots \vee \chi_k$, C' contains the constraint $p_\psi \leq p_{\chi_1} + \ldots + p_{\chi_k}$.

Both the number of constraints in C' and the number of terms in any expression that appears in one of them is bounded by a constant times the number of subformulas of φ, so the size of C is polynomial in that of φ. It remains to show that C' has property (A).

For the "if" part, let s be an arbitrary state and extend s to an assignment σ over primary variables and the secondary variables mentioned in C' by setting $\sigma[p_\psi] = 1$ if $s[\psi] = \mathit{true}$ and $\sigma[p_\psi] = 0$ if $s[\psi] = \mathit{false}$ for each subformula ψ. We will show that σ satisfies every constraint in C'. Thus, this assignment is a witness to the fact that $\{\sigma[p_\psi] > 0\} \cup C' \cup C_P(s)$ is satisfiable if $s[\psi] = \mathit{true}$.

If $\sigma[v = e] = \mathit{true}$ the constraint $v = e \rightarrow p_{v=e} = 1$ is satisfied because $\sigma[p_{v=e}] = 1$ (by construction) and the constraint $v \neq e \rightarrow p_{v=e} = 0$ is satisfied because $\sigma[v \neq e] = \mathit{false}$; if $\sigma[v = e] = \mathit{false}$, then it is the other way around. Constraints $v \neq e \rightarrow p_{v \neq e} = 1$ and $v = e \rightarrow p_{v \neq e} = 0$ are analogous. The constraints $p_\psi \leq p_{\chi_1}, \ldots, p_\psi \leq p_{\chi_k}$ created for a conjunction $\psi = \chi_1 \wedge \ldots \wedge \chi_k$ are satisfied because $\sigma[p_\psi] = 1$ only if $\sigma[\psi] = \mathit{true}$ only if $\sigma[\chi_i] = \mathit{true}$ for each conjunct χ_i, in which case $\sigma[p_{\chi_i}] = 1$; when $\sigma[p_\psi] = 0$ the constraints are satisfied because all the indicator variables are bounded be greater than or equal to zero. Similarly, the constraint $p_\psi \leq p_{\chi_1} + \ldots + p_{\chi_k}$ created for a disjunction is satisfied because $\sigma[p_\psi] = 1$ only if $\sigma[\psi] = \mathit{true}$ only if $\sigma[\chi_i] = \mathit{true}$ for at least one disjunct χ_i, in which case $\sigma[p_{\chi_i}] = 1$ which makes also the sum at least 1, and $\sigma[p_\psi] = 0$ because zero also lower-bounds the sum.

For the "only if" part, we proceed by a structural induction. As the first base case, consider an atomic subformula of the form $v = e$, and a state s such that $s(v = e) = \mathit{false}$. Then $\{\sigma(p_{v=1}) > 0\} \cup C' \cup C_P(s)$ contains $\{v = e', v \neq e \rightarrow p_{v=e} = 0, p_{v=e} > 0\}$, for some $e' \neq e$, which is clearly not satisfiable. The second base case, an atomic subformula of the form $v \neq e$, is analogous.

Consider a conjunctive formula, $\psi = \chi_1 \wedge \ldots \wedge \chi_k$. If $s[\psi] = \mathit{false}$ then $s[\chi_i] = \mathit{false}$ for at least one conjunct χ_i. By inductive assumption, this implies $\{p_{\chi_i} > 0\} \cup C' \cup C_P(s)$ is unsatisfiable. Since C' contains $p_\psi \leq p_{\chi_i}$, $p_\psi > 0$ implies $p_{\chi_i} > 0$ in any model for C', which means that $\{p_\psi > 0\} \cup C' \cup C_P(s)$ is also unsatisfiable.

Finally, consider a disjunctive formula, $\psi = \chi_1 \vee \ldots \vee \chi_k$. If $s[\psi] = \mathit{false}$ then $s[\chi_i] = \mathit{false}$ for every disjunct χ_i. By inductive assumption, this implies $\{p_{\chi_i} > 0\} \cup C' \cup C_P(s)$ is unsatisfiable. Thus, the sum $p_{\chi_1} + \ldots + p_{\chi_k}$ also cannot be greater than zero (since that would imply one of its parts is), and thus $\{p_\psi > 0\} \cup C' \cup C_P(s)$ is also unsatisfiable. $\square$

Appendix B

Domains

B.1 Min-cut

```
(define (domain min-cut)
  (:requirements :strips :typing :negative-preconditions :derived-predicates)

  (:types node edge block)

  (:predicates
   ;; static
   (adjacent ?e - edge ?f - edge)
   (edge-from ?e - edge ?n - node)
   (edge-to ?e - edge ?n - node)
   (source-node ?n - node)
   ;; derived:
   (blocked ?e - edge)
   (reachable-node ?n - node)
   (reachable-edge ?e - edge)
   (isolated ?n - node)
   )

  (:functions
   ;; primary state variables:
   (at ?b - block) - edge
   )

  (:derived (blocked ?e - edge)
            (exists (?b - block) (= (at ?b) ?e)))

  (:derived (reachable-node ?n - node) (source-node ?n))
  (:derived (reachable-node ?n - node)
            (exists (?e - edge) (and (edge-to ?e ?n)
                                     (not (blocked ?e))
                                     (reachable-edge ?e))))
  (:derived (reachable-edge ?e - edge)
            (exists (?n - node) (and (edge-from ?e ?n)
                                     (reachable-node ?n))))
  (:derived (isolated ?n - node) (not (reachable-node ?n)))

  (:action move
   :parameters (?b - block ?from - edge ?to - edge)
   :precondition (and (adjacent ?from ?to)
                      (= (at ?b) ?from))
   :effect (assign (at ?b) ?to)
   )

  )
```

B.2 Sokoban

```
(define (domain sokoban-sequential)
  (:requirements :typing :derived-predicates)
  (:types thing location direction - object
         player stone - thing)

  (:predicates
   (clear ?l - location)
   (blocked ?l - location)
   (at ?t - thing ?l - location)
   (at-goal ?s - stone)
   (IS-GOAL ?l - location)
   (IS-NONGOAL ?l - location)
   (MOVE-DIR ?from ?to - location ?dir - direction)
   (can-reach ?p - player ?l - location)
   )

  (:functions (total-cost))

  ;; axiom for at-goal
  (:derived (at-goal ?s - stone)
            (exists (?l - location) (and (is-goal ?l) (at ?s ?l))))

  ;; axiom for clear. note that only stones count as obstacles,
  ;; the player does not; he will always move so that he is not
  ;; in the way of the square he's pushing into.
  (:derived (blocked ?l - location)
            (exists (?s - stone) (at ?s ?l)))
  (:derived (clear ?l - location) (not (blocked ?l)))
  ;; (:derived (clear ?l - location)
  ;;           (forall (?s - stone) (not (at ?s ?l))))

  ;; axioms for can-reach
  (:derived (can-reach ?p - player ?l - location)
            (at ?p ?l))

  (:derived (can-reach ?p - player ?l - location)
            (and (clear ?l)
                 (exists (?d - direction ?m - location)
                         (and (MOVE-DIR ?m ?l ?d)
                              (can-reach ?p ?m)))))


  (:action push
   :parameters (?p - player ?s - stone
                ?pp1 ?ppos ?from ?to - location
                ?dir - direction)
   :precondition (and (at ?p ?pp1)
                      (can-reach ?p ?ppos)
                      (at ?s ?from)
                      (clear ?to)
```

```
                    (MOVE-DIR ?ppos ?from ?dir)
                    (MOVE-DIR ?from ?to ?dir)
                    )
  :effect (and (not (at ?p ?pp1))
               (not (at ?s ?from))
               (at ?p ?from)
               (at ?s ?to)
               (increase (total-cost) 1)
               )
  )

 )
```

B.3 Controller verification

From Ghosh et al. [66]. Sourced from `http://www.facweb.iitkgp.ernet.in/~pallab/PAPLAN.tar.gz`. The door lock system is presented here. The adaptive cruise control domain can be found from the link above.

```
(DEFINE (DOMAIN DOOR-EXAMPLE)
(:PREDICATES (ARM-AUTOLOCK) (ARM-AUTOUNLOCK) (DOORS-CLOSED) (DOORS-LOCKED)
 (KEY-IGNITION) (REMOTE-UNLOCK-CMD) (ENGINE-ON) (TRANS-MODE-DRIVE)
 (HIGH-SPEED) (LOW-SPEED) (PREV-LOW-SPEED) (STATIONARY)
 (DISABLED-CONTROL-C4-AUTO-UNLOCK) (DISABLED-CONTROL-C3-ARM-AUTO-UNLOCK)
 (DISABLED-CONTROL-C2-AUTO-LOCK) (DISABLED-CONTROL-UNMARK-PREV-LOW-SPEED)
 (DISABLED-CONTROL-MARK-PREV-LOW-SPEED) (DISABLED-CONTROL-C1-ARM-AUTO-LOCK))
(:DERIVED (DISABLED-CONTROL-C4-AUTO-UNLOCK)
 (OR (NOT (ARM-AUTOUNLOCK)) (NOT (DOORS-LOCKED))))
(:DERIVED (DISABLED-CONTROL-C3-ARM-AUTO-UNLOCK)
 (OR (NOT (REMOTE-UNLOCK-CMD))
     (NOT (DOORS-LOCKED))
     (HIGH-SPEED)
     (ARM-AUTOUNLOCK)))
(:DERIVED (DISABLED-CONTROL-C2-AUTO-LOCK)
 (OR (NOT (ARM-AUTOLOCK)) (NOT (DOORS-CLOSED)) (DOORS-LOCKED)))
(:DERIVED (DISABLED-CONTROL-UNMARK-PREV-LOW-SPEED)
 (OR (NOT (PREV-LOW-SPEED)) (LOW-SPEED) (HIGH-SPEED)))
(:DERIVED (DISABLED-CONTROL-MARK-PREV-LOW-SPEED)
 (OR (NOT (LOW-SPEED)) (PREV-LOW-SPEED)))
(:DERIVED (DISABLED-CONTROL-C1-ARM-AUTO-LOCK)
 (OR (NOT (PREV-LOW-SPEED)) (NOT (HIGH-SPEED)) (LOW-SPEED)))
(:ACTION E7-COMMAND-REMOTE-UNLOCK :PARAMETERS () :PRECONDITION
 (AND (DOORS-LOCKED)
      (NOT (REMOTE-UNLOCK-CMD))
      (DISABLED-CONTROL-C4-AUTO-UNLOCK)
      (DISABLED-CONTROL-C3-ARM-AUTO-UNLOCK)
      (DISABLED-CONTROL-C2-AUTO-LOCK)
      (DISABLED-CONTROL-UNMARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-MARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-C1-ARM-AUTO-LOCK))
 :EFFECT (AND (REMOTE-UNLOCK-CMD)))
(:ACTION E6-SPEED-LOW-TO-HIGH :PARAMETERS () :PRECONDITION
 (AND (ENGINE-ON)
      (TRANS-MODE-DRIVE)
      (LOW-SPEED)
      (DISABLED-CONTROL-C4-AUTO-UNLOCK)
      (DISABLED-CONTROL-C3-ARM-AUTO-UNLOCK)
      (DISABLED-CONTROL-C2-AUTO-LOCK)
      (DISABLED-CONTROL-UNMARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-MARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-C1-ARM-AUTO-LOCK))
 :EFFECT (AND (NOT (LOW-SPEED)) (HIGH-SPEED)))
(:ACTION E6-SPEED-STAT-TO-LOW :PARAMETERS () :PRECONDITION
 (AND (ENGINE-ON)
      (TRANS-MODE-DRIVE)
      (STATIONARY)
      (DISABLED-CONTROL-C4-AUTO-UNLOCK)
      (DISABLED-CONTROL-C3-ARM-AUTO-UNLOCK)
      (DISABLED-CONTROL-C2-AUTO-LOCK)
      (DISABLED-CONTROL-UNMARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-MARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-C1-ARM-AUTO-LOCK))
 :EFFECT (AND (NOT (STATIONARY)) (LOW-SPEED)))
(:ACTION E5-PUT-TRANSMISSION-DRIVE :PARAMETERS () :PRECONDITION
 (AND (ENGINE-ON)
      (DOORS-CLOSED)
      (NOT (TRANS-MODE-DRIVE))
      (DISABLED-CONTROL-C4-AUTO-UNLOCK)
      (DISABLED-CONTROL-C3-ARM-AUTO-UNLOCK)
      (DISABLED-CONTROL-C2-AUTO-LOCK)
      (DISABLED-CONTROL-UNMARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-MARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-C1-ARM-AUTO-LOCK))
 :EFFECT (AND (TRANS-MODE-DRIVE)))
(:ACTION E4-RUN-ENGINE :PARAMETERS () :PRECONDITION
 (AND (KEY-IGNITION)
      (NOT (ENGINE-ON))
      (DISABLED-CONTROL-C4-AUTO-UNLOCK)
      (DISABLED-CONTROL-C3-ARM-AUTO-UNLOCK)
      (DISABLED-CONTROL-C2-AUTO-LOCK)
      (DISABLED-CONTROL-UNMARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-MARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-C1-ARM-AUTO-LOCK))
 :EFFECT (AND (ENGINE-ON)))
(:ACTION E3-PUT-KEY-IN-IGNITION :PARAMETERS () :PRECONDITION
 (AND (NOT (KEY-IGNITION))
      (DISABLED-CONTROL-C4-AUTO-UNLOCK)
      (DISABLED-CONTROL-C3-ARM-AUTO-UNLOCK)
      (DISABLED-CONTROL-C2-AUTO-LOCK)
      (DISABLED-CONTROL-UNMARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-MARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-C1-ARM-AUTO-LOCK))
 :EFFECT (AND (KEY-IGNITION)))
(:ACTION E2-CLOSE-DOORS :PARAMETERS () :PRECONDITION
 (AND (NOT (DOORS-CLOSED))
      (DISABLED-CONTROL-C4-AUTO-UNLOCK)
      (DISABLED-CONTROL-C3-ARM-AUTO-UNLOCK)
      (DISABLED-CONTROL-C2-AUTO-LOCK)
      (DISABLED-CONTROL-UNMARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-MARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-C1-ARM-AUTO-LOCK))
 :EFFECT (AND (DOORS-CLOSED)))
(:ACTION E1-OPEN-DOORS :PARAMETERS () :PRECONDITION
 (AND (NOT (DOORS-LOCKED))
      (NOT (TRANS-MODE-DRIVE))
      (STATIONARY)
      (DOORS-CLOSED)
      (DISABLED-CONTROL-C4-AUTO-UNLOCK)
      (DISABLED-CONTROL-C3-ARM-AUTO-UNLOCK)
      (DISABLED-CONTROL-C2-AUTO-LOCK)
      (DISABLED-CONTROL-UNMARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-MARK-PREV-LOW-SPEED)
      (DISABLED-CONTROL-C1-ARM-AUTO-LOCK))
  :EFFECT (AND (NOT (DOORS-CLOSED))))
 (:ACTION CONTROL-C4-AUTO-UNLOCK :PARAMETERS () :PRECONDITION
  (AND (ARM-AUTOUNLOCK) (DOORS-LOCKED)) :EFFECT
  (AND (NOT (DOORS-LOCKED)) (NOT (ARM-AUTOUNLOCK))))
 (:ACTION CONTROL-C3-ARM-AUTO-UNLOCK :PARAMETERS () :PRECONDITION
  (AND (REMOTE-UNLOCK-CMD)
      (DOORS-LOCKED)
      (NOT (HIGH-SPEED))
      (NOT (ARM-AUTOUNLOCK)))
  :EFFECT (AND (ARM-AUTOUNLOCK)))
 (:ACTION CONTROL-C2-AUTO-LOCK :PARAMETERS () :PRECONDITION
  (AND (ARM-AUTOLOCK) (DOORS-CLOSED) (NOT (DOORS-LOCKED))) :EFFECT
  (AND (DOORS-LOCKED) (NOT (ARM-AUTOLOCK))))
 (:ACTION CONTROL-UNMARK-PREV-LOW-SPEED :PARAMETERS () :PRECONDITION
  (AND (PREV-LOW-SPEED) (NOT (LOW-SPEED)) (NOT (HIGH-SPEED))) :EFFECT
  (AND (NOT (PREV-LOW-SPEED))))
 (:ACTION CONTROL-MARK-PREV-LOW-SPEED :PARAMETERS () :PRECONDITION
  (AND (LOW-SPEED) (NOT (PREV-LOW-SPEED))) :EFFECT (AND (PREV-LOW-SPEED)))
 (:ACTION CONTROL-C1-ARM-AUTO-LOCK :PARAMETERS () :PRECONDITION
  (AND (PREV-LOW-SPEED) (NOT (LOW-SPEED)) (HIGH-SPEED)) :EFFECT
  (AND (ARM-AUTOLOCK) (NOT (PREV-LOW-SPEED)))))
```

B.4 Blocker

```
(define (domain blocker-strips)
  (:requirements :strips :derived-predicates)

  ;; The set of objects are the numbers from 0 to N-1, where N is the
  ;; number of nodes in the graph. These are used both to represent the
  ;; nodes in the graph and distances (because distances in the graph
  ;; will never be greater than the number of nodes minus one).

  (:predicates
   ;; primary predicates:
   (cat ?x)
   (blocked ?x)
   (blockers-turn)
   (cats-turn)
   ;; static predicates:
   (exit ?x)
   (edge ?x ?y)
   (is-zero ?x)
   (next ?x ?y)
   ;; derived predicates:
   (prefer ?x ?y)
   (cat-moves ?from ?to)
   (distance-to-exit ?x ?n)
   (closer-or-equal-to-exit ?x ?y)
   (closer-to-exit ?x ?y)
   (less ?x ?y)
   (trapped)
   )

  ;; (distance-to-exit ?x ?n) holds if we can reach an exit node from
  ;; ?x in ?n steps or less, given the current set of blocked nodes.
  (:derived (distance-to-exit ?x ?z)
          (and (exit ?x) (is-zero ?z)))

(:derived (distance-to-exit ?x ?k)
          (exists (?y ?j)
                  (and (edge ?x ?y)
                       (next ?j ?k)
                       (not (blocked ?y))
                       (distance-to-exit ?y ?j))))

(:derived (distance-to-exit ?x ?k)
          (exists (?j)
                  (and (next ?j ?k)
                       (distance-to-exit ?x ?j))))


;; (closer-to-exit ?x ?y) holds if the shortest distance to an exit
;; from ?x is stricly smaller than the shortest distance to an exit
;; from ?y.
(:derived (closer-to-exit ?x ?y)
          (exists (?k)
                  (and (distance-to-exit ?x ?k)
                       (not (distance-to-exit ?y ?k)))))

;; (closer-or-equal-to-exit ?x ?y) holds if the shortest distance to
;; an exit from ?x is less than or equal to the shortest distance to
;; an exit from ?y.
(:derived (closer-or-equal-to-exit ?x ?y)
          (not (closer-to-exit ?y ?x)))

;; (less ?x ?y) iff ?x is strictly less than ?y.
(:derived (less ?x ?y)
          (or (next ?x ?y)
              (exists (?z) (and (next ?x ?z) (less ?z ?y)))))

;; (trapped) is true iff the cat is trapped; that is, distance-to-exit
;; is false for every value from the cat's current position.
(:derived (trapped)
          (exists (?x)
                  (and (cat ?x)
                       (forall (?n) (not (distance-to-exit ?x ?n))))))

;; (prefer ?x ?y) iff cat prefers moving to ?x over ?y (only
;; relevant if ?x and ?y are both neighbours of the cat's current
;; position, but this is not tested for here). This is true if ?y
;; is blocked; ?x is strictly closer to an exit than ?y; or ?x and
;; ?y are at the same distance to exit but ?x is less than ?y (i.e.,
;; numeric order of the nodes is used as the final tie-breaker).
(:derived (prefer ?x ?y)
          (or (blocked ?y)
              (closer-to-exit ?x ?y)
              (and (closer-or-equal-to-exit ?x ?y)
                   (less ?x ?y))))

;; (cat-moves ?from ?to) iff ?to is the node that the cat will move to
;; from ?from. This is the node that is closest to an exit, or least
;; among the nodes the minimum distance to exit. If the cat is already
;; trapped, this predicate is false for all destinations.
(:derived (cat-moves ?from ?to)
          (and (edge ?from ?to)
               (not (blocked ?to))
               (not (trapped))
               (forall (?alt)
                         (or (= ?to ?alt)
                             (not (edge ?from ?alt))
                                  (prefer ?to ?alt)))))

;; In the strips formulation, the blocker's and cat's actions alternate.

;; Blocker's action:
(:action block
 :parameters (?b)
 :precondition (and (blockers-turn)
               (not (cat ?b)))
 :effect (and (blocked ?b)
              (not (blockers-turn))
              (cats-turn))
 )

;; Blocker's other action (useless):
;; (:action unblock
;;  :parameters (?b)
;;  :precondition (and (blockers-turn)
;;                     (blocked ?b))
;;  :effect (and (not (blocked ?b))
;;               (not (blockers-turn))
;;               (cats-turn))
;;  )

;; Cat's action:
(:action move
 :parameters (?from ?to)
 :precondition (and (cats-turn)
                    (cat ?from)
                    (not (exit ?from))
                    (cat-moves ?from ?to))
 :effect (and (not (cat ?from))
              (cat ?to)
              (not (cats-turn))
              (blockers-turn))
 )

)
```

B.5 Automated narrative generation

The domain:

```
(define (domain social-planning)
 (:requirements :adl :typing :derived-predicates)

 (:types locatable place - object
         character item - locatable)

 (:predicates
  ;; static properties/relations:
  (man ?c - character)
  (woman ?c - character)
  (married ?a - character ?b - character)
  (friend-of ?a - character ?b - character)
  (precious ?i - item)
  (main-character ?c - character)
 ;; character traits (also static):
 (is-greedy ?c - character)
 (is-curious ?c - character)
 (is-pursuing ?c - character)
 (is-obedient ?c - character)
 (is-suspicious ?c - character)
 (is-jealous ?c - character)
 (is-vain ?c - character)
 (is-lustful ?c - character)
 (is-wrathful ?c - character)
 (is-evil ?c - character)

 ;; state of the world:
 (alive ?c - character)
 (dead ?c - character)
 (at ?l - locatable ?p - place)
 (has ?c - character ?i - item)
 ;; state of mind:
 (loves ?c - character ?d - character)
 (believes-loves ?a - character ?c - character ?d - character)
 ;; records of passed actions:
 (requested-at ?a - character ?c - character ?p - place)
 (requested-has ?a - character ?c - character ?i - item)
 (gift ?from - character ?to - character ?i - item)
 (killed ?a - character ?c - character)

 ;; derived predicates:
 (can-see ?c - character ?l - locatable)
 (alone-at ?c - character ?p - place)
 (motive-for-has ?a - character ?c - character ?i - item)
 (motive-for-at ?a - character ?l - locatable ?p - place)
 (motive-for-dead ?a - character ?c - character)
 (reason-to-believe-loves ?a - character ?c - character ?d - character)
 (reason-to-love ?a - character ?c - character)
 )

(:derived (can-see ?c - character ?l - locatable)
          (exists (?p - place)
                  (and (at ?c ?p)
                       (at ?l ?p))))

(:derived (alone-at ?c - character ?p - place)
          (forall (?d - character)
                  (or (not (at ?d ?p))
                      (= ?d ?c))))

;; ?a has motive for (has ?a ?i) if ?a is greedy and ?i is precious:
(:derived (motive-for-has ?a - character ?a - character ?i - item)
          (and (is-greedy ?a)
               (precious ?i)
               (can-see ?a ?i)))

;; ?a has motive for (at ?a ?p) if ?a is curiuous and a friend of
;; ?a has asked ?a to be at ?p:
(:derived (motive-for-at ?a - character ?a - character ?p - place)
          (and (is-curious ?a)
               (exists (?f - character)
                       (and (friend-of ?a ?f)
                            (requested-at ?f ?a ?p)))))

;; ?a has motive for (has ?c ?i) if ?a is "pursuing", in love
;; with ?c, and ?i is precious:
```

```
(:derived (motive-for-has ?a - character ?c - character ?i - item)
          (and (is-pursuing ?a)
               (loves ?a ?c)
               (precious ?i)))

;; ?a has motive for (has ?c ?i) if ?a is obedient, loves ?c,
;; and ?c has requested to have ?i
(:derived (motive-for-has ?a - character ?c - character ?i - item)
          (and (is-obedient ?a)
               (loves ?a ?c)
               (requested-has ?c ?c ?i)))

;; ?a has motive for (has ?a ?i) if ?a has motive for (has ?c ?i):
(:derived (motive-for-has ?a - character ?a - character ?i - item)
          (exists (?c - character)
                  (motive-for-has ?a ?c ?i)))

;; ?a has motive for (at ?a ?p) if ?a has motive for (has ?c ?i)
;; ?a has ?i, and ?c is at ?p:
(:derived (motive-for-at ?a - character ?a - character ?p - place)
          (exists (?c - character ?i - item)
                  (and (motive-for-has ?a ?c ?i)
                       (has ?a ?i)
                       (at ?c ?p))))

;; ?a has motive for (at ?a ?p) if ?a has motive for (has ?a ?i)
;; and ?i is at ?p:
(:derived (motive-for-at ?a - character ?a - character ?p - place)
          (exists (?i - item)
                  (and (motive-for-has ?a ?a ?i)
                       (at ?i ?p))))

;; ?a has motive for (at ?c ?p) if ?a has motive for (has ?a ?i)
;; and ?c is at ?q and ?i is at ?q and ?p != ?q (i.e., ?a is
;; planning to steal ?i at ?p):
(:derived (motive-for-at ?a - character ?c - character ?p - place)
          (exists (?i - item ?q - place)
                  (and (motive-for-has ?a ?a ?i)
                       (at ?i ?q)
                       (at ?c ?q))))

;; ?a has motive for (dead ?c) if ?a is jealous, ?c is the spouse of ?a,
;; and ?a believes ?c loves ?d:
(:derived (motive-for-dead ?a - character ?c - character)
          (and (is-jealous ?a)
               (not (= ?c ?a))
               (exists (?d - character)
                       (and (married ?a ?c)
                            (believes-loves ?a ?c ?d)))))

;; ?a has motive for (dead ?d) if ?a is jealous, ?c is the spouse of ?a,
;; and ?a believes ?c loves ?d:
(:derived (motive-for-dead ?a - character ?d - character)
          (and (is-jealous ?a)
               (not (= ?d ?a))
               (exists (?c - character)
                       (and (married ?a ?c)
                            (believes-loves ?a ?c ?d)))))

;; ?a has motive for (dead ?c) if ?a is wrathful, ?a loves ?d, and
;; ?c has killed ?d:
(:derived (motive-for-dead ?a - character ?c - character)
          (and (is-wrathful ?a)
               (not (= ?c ?a))
               (exists (?d - character)
                       (and (loves ?a ?d)
                            (killed ?c ?d)))))

;; ?a has motive for (at ?a ?p) if ?a has motive for (dead ?c)
;; and ?c is at ?p:
(:derived (motive-for-at ?a - character ?a - character ?p - place)
          (exists (?c - character)
                  (and (motive-for-dead ?a ?c)
                       (at ?c ?p))))

;; Belief revision rules:

;; ?a may (believe-loves ?a ?c ?d) if ?a is suspicious, and ?a observes
;; (has ?d ?i) for an ?i that was a gift from ?a to ?c
(:derived (reason-to-believe-loves
           ?a - character ?c - character ?d - character)
          (and (is-suspicious ?a)
               (not (= ?c ?d))
               (not (= ?a ?c))
               (not (= ?a ?d))
               (exists (?i - item)
                       (and (gift ?a ?c ?i)
                            (has ?d ?i)
                            (can-see ?a ?d)))))

;; ?a may fall in love with ?c if ?a is a woman, ?a is vain,
;; ?c is a man, and ?c has given ?a something precious:
(:derived (reason-to-love ?a - character ?c - character)
          (and (woman ?a)
               (is-vain ?a)
               (man ?c)
               (exists (?i - item)
                       (and (gift ?c ?a ?i)
                            (precious ?i)))))

;; ?a may fall in love with ?c if ?a is a man, ?a is "lustful",
;; ?c is a woman, and ?c is wearing ("has") something precious:
(:derived (reason-to-love ?a - character ?c - character)
          (and (man ?a)
               (is-lustful ?a)
               (woman ?c)
               (can-see ?a ?c)
               (exists (?i - item)
                       (and (has ?c ?i)
                            (precious ?i)))))

;; Belief revision actions:
;; Beliefs can persist beyond the current state, so we need to have
;; actions that allow characters to adopt beliefs.

(:action adopt-belief-loves
 :parameters (?a - character ?c - character ?d - character)
 :precondition (and (reason-to-believe-loves ?a ?c ?d)
                    (alive ?a)
                    (alive ?c))
 :effect (believes-loves ?a ?c ?d)
 )

(:action fall-in-love
 :parameters (?a - character ?c - character)
 :precondition (and (reason-to-love ?a ?c)
                    (not (= ?a ?c))
                    (alive ?a)
                   (alive ?c))
 :effect (loves ?a ?c)
 )

;; Character actions:

(:action take
 :parameters (?a - character ?i - item ?p - place)
 :precondition (and (or (main-character ?a)
                        (motive-for-has ?a ?a ?i))
                    (alive ?a)
                    (at ?i ?p)
                    (at ?a ?p)
                    (alone-at ?a ?p))
 :effect (and (not (at ?i ?p))
              (has ?a ?i))
 )

(:action drop
 :parameters (?a - character ?i - item ?p - place)
 :precondition (and (or (main-character ?a)
                        (motive-for-at ?a ?i ?p))
                    (alive ?a)
                    (has ?a ?i)
                    (at ?a ?p))
 :effect (and (not (has ?a ?i))
              (at ?i ?p))
 )

(:action give
 :parameters (?a - character ?c - character ?i - item ?p - place)
 :precondition (and (or (main-character ?a)
                        (motive-for-has ?a ?c ?i))
                    (alive ?a)
                    (alive ?c)
                    (has ?a ?i)
                    (at ?a ?p)
                    (at ?c ?p))
 :effect (and (not (has ?a ?i))
              (has ?c ?i)
              (gift ?a ?c ?i))
 )

(:action goto
 :parameters (?a - character ?p - place ?from - place)
 :precondition (and (or (main-character ?a)
                        (motive-for-at ?a ?a ?p))
                    (alive ?a)
                    (at ?a ?from)
                    (not (= ?p ?from)))
 :effect (and (not (at ?a ?from))
              (at ?a ?p))
 )

;; Although he is evil, Iago is too cautious to kill other
;; characters himself...
(:action kill
 :parameters (?a - character ?c - character ?p - place)
```

```
   :precondition (and (motive-for-dead ?a ?c)
                      (alive ?a)
                      (alive ?c)
                      (at ?a ?p)
                      (at ?c ?p))
   :effect (and (not (alive ?c))
                (dead ?c)
                (killed ?a ?c))
   )

  ;; Communicative actions. Note that requests (as modelled here) are
  ;; not made to a specific character.

  (:action request-at
   :parameters (?a - character ?c - character ?p - place)
   :precondition (and (or (main-character ?a)
                          (motive-for-at ?a ?c ?p))
                      (alive ?a)
                      (alive ?c))
   :effect (requested-at ?a ?c ?p)
   )

  (:action request-has
   :parameters (?a - character ?c - character ?i - item)
   :precondition (and (or (main-character ?a)
                          (motive-for-has ?a ?c ?i))
                      (alive ?a)
                      (alive ?c))
   :effect (requested-has ?a ?c ?i)
   )

  )
```

An instance (corresponding to Iago's problem):

```
;; An instance of the social planning domain, based on Iago's problem:
;; How to make Othello kill Desdemona and then die?

(define (problem Iago-2)
  (:domain social-planning)

  (:objects
   Iago Othello Emilia Desdemona Cassio - character
   handkerchief - item
   garden bedroom residence palace - place
   )

  (:init
   (man Iago)
   (man Othello)
   (woman Emilia)
   (woman Desdemona)
   (man Cassio)
   (married Iago Emilia)
   (married Emilia Iago)
   (married Othello Desdemona)
   (married Desdemona Othello)
   (friend-of Cassio Iago)
   (friend-of Othello Iago)
   (friend-of Desdemona Emilia)
  (precious handkerchief)
  (loves Emilia Iago)
  (loves Othello Desdemona)
  (loves Desdemona Othello)
  (is-curious Othello)
  (is-jealous Othello)
  (is-suspicious Othello)
  (is-wrathful Othello)
  (is-curious Cassio)
  (is-greedy Cassio)
  (is-lustful Cassio)
  (is-wrathful Cassio)
  (is-obedient Emilia)
  (is-vain Emilia)
  (is-curious Desdemona)
  (is-obedient Desdemona)
  (is-evil Iago) ;; MUHAHAHAHA!!
  (main-character Iago)
  (alive Iago)
  (alive Othello)
  (alive Emilia)
  (alive Desdemona)
  (alive Cassio)
  (at Iago garden)
  (at Othello palace)
  (at Desdemona bedroom)
  (at Emilia garden)
  (at Cassio residence)
  (at handkerchief bedroom)
  (gift Othello Desdemona handkerchief)
  )

 (:goal (and (killed Othello Desdemona)
             (dead Othello)))

 )
```

Bibliography

[1] Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[2] Ruth Aylett, James K. Soutter, Gary J. Petley, and Paul W. H. Chung. AI planning in a chemical plant domain. In *ECAI*, pages 622–626, 1998.

[3] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artif. Intell.*, 116(1-2):123–191, 2000.

[4] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, pages 268–283, 2001.

[5] ME Baran and Felix F Wu. Optimal sizing of capacitors placed on a radial distribution system. *IEEE Transactions on power Delivery*, 4(1):735–743, 1989.

[6] A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, Y. Sun, and D. Weld. UCPOP user's manual. Technical Report 93-09-06d, University of Washington, CS Dept., 1995.

[7] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[8] Piergiorgio Bertoli, Alessandro Cimatti, Marco Roveri, and Paolo Traverso. Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of the Seventeenth*

International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001, pages 473–478, 2001.

[9] Christoph Betz and Malte Helmert. Planning with h^+ in theory and practice. In *KI 2009: Advances in Artificial Intelligence, 32nd Annual German Conference on AI, Paderborn, Germany, September 15-18, 2009. Proceedings*, pages 9–16, 2009.

[10] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pages 1636–1642, 1995.

[11] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):281 – 300, 1997.

[12] Mark S. Boddy and Daniel P. Johnson. A new method for the global solution of large systems of continuous constraints. In *Global Optimization and Constraint Satisfaction, First International Workshop Global Constraint Optimization and Constraint Satisfaction, COCOS 2002, Valbonne-Sophia Antipolis, France, October 2-4, 2002, Revised Selected Papers*, pages 142–156, 2002.

[13] B. Bonet and H. Geffner. GPT: a tool for planning with uncertainty and partial information. In *IJCAI Workshop on Planning under Uncertainty and Incomplete Information*, pages 82–87, 2001.

[14] B. Bonet and M. Helmert. Strengthening landmark heuristics via hitting sets. In *ECAI*, pages 329–334, 2010.

[15] Blai Bonet and Hector Geffner. Planning with incomplete information as heuristic search in belief space. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14-17, 2000*, pages 52–61, 2000.

[16] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[17] Blai Bonet, Gábor Loerincs, and Hector Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island.*, pages 714–719, 1997.

[18] Blai Bonet and Sylvie Thiébaux. GPT meets PSR. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003), June 9-13, 2003, Trento, Italy*, pages 102–112, 2003.

[19] Ronen I. Brafman and Guy Shani. A multi-path compilation approach to contingent planning. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, 2012.

[20] Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69:165–204, 1994.

[21] Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *COMPUTATIONAL INTELLIGENCE*, 11:625–655, 1993.

[22] Anthony R. Cassandra, Leslie Pack Kaelbling, and Michael L. Littman. Acting optimally in partially observable stochastic domains. In *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 2.*, pages 1023–1028, 1994.

[23] P. P. Chakrabarti, Sujoy Ghose, Arup Acharya, and S. C. De Sarkar. Heuristic search in restricted memory. *Artif. Intell.*, 41(2):197–221, 1989.

[24] Hsueh-Min Chang and Von-Wun Soo. Simulation-based story generation with a theory of mind. In *Proc. Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, pages 16–21, 2008.

[25] Gianfranco Ciardo and Radu Siminiceanu. Using edge-valued decision diagrams for symbolic generation of shortest paths. In *Formal Methods in Computer-Aided Design, 4th International Conference, FMCAD 2002, Portland, OR, USA, November 6-8, 2002, Proceedings*, pages 256–273, 2002.

[26] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, pages 495–499, 1999.

[27] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169, 2000.

[28] Carleton Coffrin and Pascal Van Hentenryck. A linear-programming approximation of AC power flows. *INFORMS Journal on Computing*, 26(4):718–734, 2014.

[29] A. J. Coles, A. I. Coles, M. Fox, and D. Long. Forward-chaining partial-order planning. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS-10)*, May 2010.

[30] Amanda Jane Coles, Andrew Coles, Maria Fox, and Derek Long. Colin: Planning with continuous linear numeric change. *CoRR*, abs/1401.5857, 2014.

[31] Andrew Coles, Maria Fox, Keith Halsey, Derek Long, and Amanda Smith. Managing concurrency in temporal planning using planner-scheduler interaction. *Artif. Intell.*, 173(1):1–44, 2009.

[32] Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. A hybrid relaxed planning graph-lp heuristic for numeric planning domains, 2008.

[33] Andrew Coles, Maria Fox, Derek Long, and Amanda Smith. Planning with problems requiring temporal coordination. In *AAAI*, pages 892–897, 2008.

[34] Andrew Coles and Amanda Smith. Marvin: A heuristic search planner with online macro-action learning. *JAIR*, 28:119–156, 2007.

[35] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[36] Marina Davidson and Max Garagnani. Pre-processing planning domains containing language axioms. In *In Proc. UK PlanSIG workshop*, pages 23–34, 2002.

[37] Thomas L. Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann E. Nicholson. Planning with deadlines in stochastic domains. In *Proceedings of the 11th National Conference on Artificial Intelligence. Washington, DC, USA, July 11-15, 1993.*, pages 574–579, 1993.

[38] Patrick Doherty and Jonas Kvarnström. Talplanner: A temporal logic-based planner. *AI Magazine*, 22(3):95–102, 2001.

[39] Carmel Domshlak and Adeline Nazarenko. The complexity of optimal monotonic planning: The bad, the good, and the causal graph. *J. Artif. Intell. Res. (JAIR)*, 48:783–812, 2013.

[40] Christian Dornhege, Patrick Eyerich, Thomas Keller, Sebastian Trüg, Michael Brenner, and Bernhard Nebel. Semantic attachments for domain-independent planning systems. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, 2009.

[41] S. Edelkamp. Planning with pattern databases. In *Proc. 6th European Conference on Planning (ECP)*, pages 13–24, 2001.

[42] S. Edelkamp. Optimal symbolic PDDL3 planning with MIPS-BDD. In *5th International Planning Competition Booklet*, 2006.

[43] Stefan Edelkamp and Jörg Hoffmann. The deterministic part of IPC-4: an overview. *CoRR*, abs/1109.5663, 2011.

[44] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, 2009.

[45] R. Fagin, J.Y. Halpern, Y. Moses, and M. Vardi. *Reasoning About Knowledge.* MIT Press, 2004.

[46] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge.* MIT Press, Cambridge, MA, USA, 2003.

[47] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. Sturtevant, J. Schaeffer, and R. Holte. Partial-expansion A* with selective node generation. In *AAAI*, pages 471–477, 2012.

[48] Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell.*, 2(3/4):189–208, 1971.

[49] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR*, 20:61–124, 2003.

[50] Maria Fox and Derek Long. Pddl+: Modelling continuous time-dependent effects.

[51] G. Francès and H. Geffner. Modelling and computation in planning: Better heuristics from more expressive languages. In *Proc. 25th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 70–78, 2015.

[52] Massimiliano Garagnani. A correct algorithm for efficient planning with preprocessed domain axioms, 2000.

[53] B. Cenk Gazen and Craig A. Knoblock. Combining the expressivity of ucpop with the efficiency of graphplan. In Sam Steel and Rachid Alami, editors, *ECP*, volume 1348 of *Lecture Notes in Computer Science*, pages 221–233. Springer, 1997.

[54] M. Gebser, B. Kaufmann, and T. Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012.

[55] Avitan Gefen and Ronen I. Brafman. The minimal seed set problem. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*, 2011.

[56] Avitan Gefen and Ronen I. Brafman. Pruning methods for optimal delete-free planning. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, 2012.

[57] H. Geffner and B. Bonet. *A Concise Introduction to Models and Methods for Automated Planning.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013.

[58] Florian Geißer, Thomas Keller, and Robert Mattmüller. Delete relaxations for planning with state-dependent action costs. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1573–1579, 2015.

[59] Florian Geißer, Thomas Keller, and Robert Mattmüller. Delete relaxations for planning with state-dependent action costs. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel.*, pages 228–229, 2015.

[60] Florian Geißer, Thomas Keller, and Robert Mattmüller. Abstractions for planning with state-dependent action costs. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016.*, pages 140–148, 2016.

[61] Michael Gelfond. Answer sets. In *Handbook of Knowledge Representation*, pages 285–316. Elsevier, 2008.

[62] A. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artif. Intell.*, 173(5-6):619–668, 2009.

[63] Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. An approach to temporal planning and scheduling in domains with predictable exogenous events. *J. Artif. Intell. Res.*, 25:187–231, 2006.

[64] Alfonso Gerevini, Alessandro Saetti, Ivan Serina, and Paolo Toninelli. Fast planning in domains with derived predicates: An approach based on rule-action graphs and local search. In *Proc. AAAI Conference*, pages 1157–1162, 2005.

[65] Malik Ghallab and Hervé Laruelle. Representation and control in ixtet, a temporal planner. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems, University of Chicago, Chicago, Illinois, USA, June 13-15, 1994*, pages 61–67, 1994.

[66] Kamalesh Ghosh, Pallab Dasgupta, and S. Ramesh. Automated planning as an early verification tool for distributed control. *Journal of Automated Reasoning*, 54:31–68, 2015.

[67] Robert P. Goldman and Mark S. Boddy. Expressive planning and explicit knowledge. In *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, May 29-31, 1996*, pages 110–117, 1996.

[68] P. Gregory, D. Long, M. Fox, and C. Beck. Planning modulo theories: Extending the planning paradigm. In *Proc. 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, pages 65–73, 2012.

[69] P. Haslum. Optimal delete-relaxed (and semi-relaxed) planning with conditional effects. In *IJCAI*, pages 2291–2297, 2013.

[70] P. Haslum, B. Bonet, and H. Geffner. New admissible heuristics for domain-independent planning. In *Proc. AAAI Conference*, pages 1163–1168, 2005.

[71] Patrik Haslum. *Admissible Heuristics for Automated Planning.* PhD thesis, Linköping UniversityLinköping University, KPLAB - Knowledge Processing Lab, The Institute of Technology, 2006.

[72] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pages 1007–1012, 2007.

[73] Patrik Haslum, Franc Ivankovic, Miquel Ramirez, Dan Gordon, Sylvie Thiébaux, Vikas Shishankar, and Dana S. Nau. Extending classical planning with state constraints: Heuristics and search for optimal planning. *J. Artif. Intell. Res. (JAIR)*, 2018.

[74] Patrik Haslum, John K. Slaney, and Sylvie Thiébaux. Minimal landmarks for optimal delete-free planning. In *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*, 2012.

[75] Matthew Hatem and Wheeler Ruml. Simpler bounded suboptimal search. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 856–862, 2014.

[76] D. Hazarika and A.K. Sinha. An algorithm for standing phase angle reduction for power system restoration. *IEEE Trans. Power Systems*, 14(4):1213–1218, 1999.

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

[78] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5-6):503–535, 2009.

[79] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*, 2009.

[80] Malte Helmert, Patrik Haslum, Jörg Hoffmann, and Raz Nissim. Merge-and-shrink abstraction: A method for generating lower bounds in factored state spaces. *J. ACM*, 61(3):16:1–16:63, June 2014.

[81] J. Hoffmann. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *ISMIS*, pages 216–227, 2000.

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

[83] Jörg Hoffmann. The metric-ff planning system: Translating ”ignoring delete lists” to numeric state variables. *J. Artif. Int. Res.*, 20(1):291–341, December 2003.

[84] Jörg Hoffmann and Ronen I. Brafman. Contingent planning via heuristic forward search witn implicit belief states. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*, pages 71–80, 2005.

[85] Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *J. Artif. Int. Res.*, 14(1):253–302, May 2001.

[86] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[87] Tatsuya Imai and Alex Fukunaga. On a practical, integer-linear programming model for delete-free tasks and its use as a heuristic for cost-optimal planning. *J. Artif. Intell. Res. (JAIR)*, 54:631–677, 2015.

[88] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Partially observable markov decision processes for artificial intelligence. In *KI-95: Advances in Artificial Intelligence, 19th Annual German Conference on Artificial Intelligence, Bielefeld, Germany, September 11-13, 1995, Proceedings*, pages 1–17, 1995.

[89] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134, 1998.

[90] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[91] E. Karpas and C. Domshlak. Cost-optimal planning with landmarks. In *IJCAI*, 2009.

[92] Henry A. Kautz and Joachim P. Walser. State-space planning by integer optimization. In *Proceedings of the Sixteenth National Conference*

on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA., pages 526–533, 1999.

[93] L. G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096, 1979.

[94] Philip Kilby, Ignasi Abío, Daniel Guimarans, Daniel Harabor, Patrick Haslum, Valentin Mayer-Eichberger, Fazlul Siddiqui, Sylvie Thiébaux, and Tomasso Urli. There's more than one way to solve a long-haul transportation problem. In *Vehicle Routing and Logistics 2015*, VeRoLog 2015, 2015.

[95] F. Kominis and H. Geffner. Beliefs in multiagent planning: From one agent to many. In *Proc. 25th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 147–155, 2015.

[96] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell.*, 27(1):97–109, 1985.

[97] Richard E. Korf. Linear-space best-first search. *Artif. Intell.*, 62(1):41–78, 1993.

[98] Richard E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island.*, pages 700–705, 1997.

[99] Richard E. Korf and Weixiong Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA.*, pages 910–916, 2000.

[100] Matt Kraning, Eric Chu, Javad Lavaei, and Stephen Boyd. Dynamic network energy management via proximal message passing. 2013.

[101] Yung-Te Lai, Massoud Pedram, and Sarma B. K. Vrudhula. Formal verification using edge-valued binary decision diagrams. *IEEE Trans. Computers*, 45(2):247–255, 1996.

[102] Hui X. Li and Brian C. Williams. Generative planning for hybrid systems based on flow tubes. In Jussi Rintanen, Bernhard Nebel,

J. Christopher Beck, and Eric A. Hansen, editors, *ICAPS*, pages 206–213. AAAI, 2008.

[103] Johannes Löhr, Patrick Eyerich, Thomas Keller, and Bernhard Nebel. A planning based framework for controlling hybrid systems, 2012.

[104] Zohar Manna and Richard J. Waldinger. How to clear a block: A theory of plans. *J. Autom. Reasoning*, 3(4):343–377, 1987.

[105] D. Mcdermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl - the planning domain definition language. Technical Report TR-98-003, Yale Center for Computational Vision and Control,, 1998.

[106] Vitaly Mirkis and Carmel Domshlak. Cost-sharing approximations for h+. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, pages 240–247, 2007.

[107] Dana Nau, Malik Ghallab, and Paolo Traverso. *Automated Planning: Theory & Practice.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[108] Dana S. Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. SHOP: simple hierarchical ordered planner. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 968–975, 1999.

[109] B. Nebel. On the compilability and expressive power of propositional planning formalisms. *Journal of AI Research*, 12:271–315, 2000.

[110] Allen Newell and H. A. Simon. Computers &amp; thought. chapter GPS, a Program That Simulates Human Thought, pages 279–293. MIT Press, Cambridge, MA, USA, 1995.

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

[112] Héctor Palacios and Hector Geffner. Compiling uncertainty away in conformant planning problems with bounded width. *J. Artif. Intell. Res. (JAIR)*, 35:623–675, 2009.

[113] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92). Cambridge, MA, October 25-29, 1992.*, pages 103–114, 1992.

[114] J. Scott Penberthy and Daniel S. Weld. Temporal planning with continuous change. In *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 2.*, pages 1010–1015, 1994.

[115] Chiara Piacentini, Varvara Alimisis, Maria Fox, and Derek Long. Combining a temporal planner with an external solver for the power balancing problem in an electricity network. In *ICAPS*, 2013.

[116] Ira Pohl. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 12–17, 1973.

[117] F. Pommerening and M. Helmert. Optimal planning for delete-free tasks with incremental LM-cut. In *ICAPS*, 2012.

[118] L. Powell. *Power System Load Flow Analysis.* McGraw-Hill Education, 2004.

[119] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1):81 – 132, 1980.

[120] S. Richter and M. Helmert. Preferred operators and deferred evaluation in satisficing planning. In *ICAPS*, pages 273–280, 2009.

[121] Silvia Richter and Matthias Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *JAIR*, 39:127–177, 2010.

[122] Jussi Rintanen. Engineering efficient planners with SAT. In *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, pages 684–689, 2012.

[123] Nathan Robinson, Sheila A. McIlraith, and David Toman. Cost-based query optimization via AI planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2344–2351, 2014.

[124] Stéphane Ross, Joelle Pineau, Sébastien Paquet, and Brahim Chaib-draa. Online planning algorithms for pomdps. *CoRR*, abs/1401.3436, 2014.

[125] Stuart J. Russell. Efficient memory-bounded search methods. In *ECAI*, pages 1–5, 1992.

[126] Enrico Scala, Miquel Ramírez, Patrik Haslum, and Sylvie Thiébaux. Numeric planning with disjunctive global constraints via SMT. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016.*, pages 276–284, 2016.

[127] F. C. Schweppe and D. B. Rom. Power System Static-State Estimation, Part II: Approximate Model. *power apparatus and systems, ieee transactions on*, PAS-89(1):125–130, 1970.

[128] Jendrik Seipp and Malte Helmert. Counterexample-guided cartesian abstraction refinement. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*, 2013.

[129] Ji-Ae Shin and Ernest Davis. Processes and continuous change in a sat-based planner. *Artif. Intell.*, 166(1-2):194–253, August 2005.

[130] John Slaney and Sylvie Thiébaux. Blocks world revisited. *Artificial Intelligence*, 125(1):119 – 153, 2001.

[131] David E. Smith and Daniel S. Weld. Conformant graphplan. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, pages 889–896, 1998.

[132] Brian Stott, Jorge Jardim, and Ongun Alsaç. Dc power flow revisited. *IEEE Transactions on Power Systems*, 24(3):1290–1300, 2009.

[133] Jordan Tyler Thayer and Wheeler Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, pages 674–679, 2011.

[134] Sylvie Thiébaux, Carleton Coffrin, Hassan Hijazi, and John K. Slaney. Planning with MIP for supply restoration in power distribution systems. In *IJCAI*, 2013.

[135] Sylvie Thiébaux and Marie-Odile Cordier. Supply restoration in power distribution systems – a benchmark for planning under uncertainty. In *ECP*, 2001.

[136] Sylvie Thiébaux, Jörg Hoffmann, and Bernhard Nebel. In defense of PDDL axioms. *Artif. Intell.*, 168(1-2):38–69, 2005.

[137] K. Tierney, A. Coles, A. Coles, Kroer C., A.M. Britt, and R.M. Jensen. Automated planning for liner shipping fleet repositioning. In *ICAPS*, pages 279–287, 2012.

[138] H. van Ditmarsch, W. van der Hoek, and B. Kooi. *Dynamic Epistemic Logic*. Synthese Library. Springer Netherlands, 2007.

[139] Thomas Vossen, Michael O. Ball, Amnon Lotem, and Dana S. Nau. On the use of integer programming models in AI planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 304–309, 1999.

[140] Daniel Weld and Oren Etzioni. The first law of robotics (a call to arms).

[141] Terry Winograd. *Understanding Natural Language*. Academic Press, Inc., Orlando, FL, USA, 1972.

[142] Steven A. Wolfman and Daniel S. Weld. The LPSAT engine & its application to resource planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 99, Stockholm, Sweden, July 31 - August 6, 1999. 2 Volumes, 1450 pages*, pages 310–317, 1999.

[143] T. Yoshizumi, T. Miura, and T. Ishida. A* with partial expansion for large branching factor problems. In *AAAI*, 2000.