Effective Planning with Expressive Languages

Guillem Francès Medina

TESI DOCTORAL UPF / 2017

Director de la tesi

Prof. Héctor Geffner
Department of Information and Communication Technologies

The court's PhD was appointed by the rector of the Universitat Pompeu Fabra on ....., 2017.

Chairman

Member

Member

Member

Secretary

The doctoral defense was held on ....., 2017, at the Universitat Pompeu Fabra and scored as .....

PRESIDENT MEMBERS

SECRETARY

To my family, in the broadest possible sense.

En aquel Imperio, el Arte de la Cartografía logró tal Perfección que el mapa de una sola Provincia ocupaba toda una Ciudad, y el mapa del Imperio, toda una Provincia. Con el tiempo, estos Mapas Desmesurados no satisficieron y los Colegios de Cartógrafos levantaron un Mapa del Imperio, que tenía el tamaño del Imperio y coincidía puntualmente con él.

Menos Adictas al Estudio de la Cartografía, las Generaciones Siguientes entendieron que ese dilatado Mapa era Inútil y no sin Impiedad lo entregaron a las Inclemencias del Sol y los Inviernos. En los desiertos del Oeste perduran despedazadas Ruinas del Mapa, habitadas por Animales y por Mendigos; en todo el País no hay otra reliquia de las Disciplinas Geográficas.

Suárez Miranda, Viajes de Varones Prudentes, Libro Cuarto, Cap. XLV, Lérida, 1658.

⋆ ⋆ ⋆

Jorge Luis Borges, *Del Rigor en la Ciencia.*

In that Empire, the Art of Cartography attained such Perfection that the map of a single Province occupied the entirety of a City, and the map of the Empire, the entirety of a Province. In time, those Unconscionable Maps no longer satisfied, and the Cartographers Guilds struck a Map of the Empire whose size was that of the Empire, and which coincided point for point with it.

The following Generations, who were not so fond of the Study of Cartography as their Forebears had been, saw that that vast map was Useless, and not without some Pitilessness was it, that they delivered it up to the Inclemencies of Sun and Winters. In the Deserts of the West, still today, there are Tattered Ruins of that Map, inhabited by Animals and Beggars; in all the Land there is no other Relic of the Disciplines of Geography.

Suárez Miranda, Travels of Prudent Men, Book Four, Ch. XLV, Lérida, 1658.

⋆ ⋆ ⋆

Jorge Luis Borges, *On Rigor in Science.*

Acknowledgements

This thesis is the result of four years of challenging work, which would no doubt have been way more challenging, if not hopeless, without the help and support of many people. Let me start by acknowledging all of the *scientific* support. I am hugely indebted to my advisor and friend Héctor Geffner for being an inspiration and a model during all of these years. I am indebted too to each of my other coauthors, whose work and collaboration has had a direct impact on this thesis: Jonathan Ferrer, Nir Lipovetzky and Miquel Ramírez. I have also been lucky enough to share the time and knowledge of many of the researchers which have worked in the Artificial Intelligence group here at the Universitat Pompeu Fabra, in Barcelona, during these years. This includes Alex Albore, Héctor Palacios, Anders Jonsson, Blai Bonet, Jorge Lobo, Víctor Dalmau, Vicenç Gómez, Gergely Neu, Sergio Jiménez and Dimitri Ognibene plus, of course, my fellow PhD students: Jonathan Ferrer, Filippos Kominis, Damir Lotinac, Javi Segovia, Oussam Larkem and Miquel Junyent. The endless support from Lydia García and her coworkers at the Department Administration has been invaluable for the successful completion of my PhD.

I would also like to acknowledge the collaborative and truly positive environment made possible by everyone at the Simulpast research project, particularly Marco Madella, Carla Lancelotti and Xavi Rubio. I am also grateful to my Master thesis advisor, Carme Àlvarez, for her support and teachings, which I fondly remember in spite of all the years since. Special thanks go to everyone that made my research stay in Melbourne possible, particularly to Peter Stuckey and Adrian Pearce, and of course to Nir, Miquel, Sergio and Angela, who not only made it possible but also extremely enjoyable.

Even more important than scientific support is however love and friendship, as there is more to life than work, and work is meaningless without that other life — or, paraphrasing Kundera, *life is elsewhere*. I shall of course begin by thanking my parents, who have done so much for me over the years, for their love and encouragement, a gratitude which extends to my grandparents and the rest of my family. All of these years have been extremely enjoyable, in spite of clichés, thanks to all of them, and to all of the people with whom I have had the chance to live and to share my life. This includes an extraordinary (and extraordinarily large!) group of flatmates:

Sandra, Virgi, Jose, Diana, Marise, Giorgio, Sara, Cris, Shere, and María as well as the amazing network of people (or should I call them *family*?) that has grown around that place we have called our home for all of these years, which is definitely too large to enumerate, but of which I have to mention Jordi, Gael, Joana, Pol, Nati, Anuar, Montse, Carlitos, Jota and Ceci. Besides them, thanks go to so many friends over the years for their love and support: to Mireia, Xiana, Clarissa, Marion, Sylvie, David, Cris, Marc, Miriam, Jordi, Clara, Bea, Milaine and, last but not least, to Guillermo, Sandra and Shere, for all the things we have lived together, for all the ones yet to be lived. To all of them goes my gratitude. *We carry a new world here in our hearts.*

Abstract

Classical planning is concerned with finding sequences of actions that achieve a certain goal from an initial state of the world, assuming that actions are deterministic, states are fully known, and both are described in some modeling language. This work develops effective means of dealing with expressive modeling languages for classical planning. First, we show that expressive languages not only allow simpler problem representations, but also capture additional problem structure that can be leveraged by heuristic solution methods. We develop heuristics that support functions and existential quantification in the problem definition, and show empirically that they can be more informed and cost-effective. Second, we develop a novel width-based algorithm that matches state-of-the-art performance without looking at the declarative representation of actions. This is a significant departure from previous research, and advances the use of expressive modeling languages in planning and the scope and effectiveness of classical planners.

Resum

La planificació clàssica consisteix en trobar una seqüència d'accions que meni d'un cert estat inicial fins a un estat desitjat, on les accions són deterministes, els estats perfectament coneguts, i ambdós elements són descrits en algun llenguatge formal. En aquest treball desenvolupem mitjans efectius de tractar amb llenguatges expressius de planificació clàssica. Primer, mostrem que un llenguatge més expressiu no només permet obtenir representacions compactes, sinó que permet capturar també estructura del problema aprofitable mitjançant mètodes heuristics, desenvolupem heurístiques que suporten funcions i quantificació existencial en la definició del problema, i demostrem empíricament que poden ser més informades i efectives. En segon lloc, desenvolupem un nou algorisme que ofereix rendiment similar a l'estat de l'art sense necessitat de cap representació declarativa de les accions. Això suposa una innovació significativa respecte a la recerca anterior, i un avenç en l'ús de llenguatges expressius i en l'abast i efectivitat dels planificadors clàssics.

Preface

All models are wrong, but some are useful.

George Box

One of the foundational cornerstones of Artificial Intelligence, planning is the model-based approach to intelligent behavior, where a model of the world and of possible actions to be taken is used to decide on a course of action that brings the world to some desired state. Such a model of world and actions needs to be represented in some modeling language. Modeling languages aim at satisfying two conflicting goals: on the one hand, they need to be expressive enough so that compact and natural problem representations are possible; on the other hand, they need to be simple enough so that *general* computational methods can be developed to solve in an *efficient* manner any problem that can be represented in the language. Unfortunately, no modeling language is known in classical planning which can represent a wide range of interesting problems and is computationally tractable at the same time, something which is not exclusive of planning and is more generally referred to as the tradeoff between expressiveness and tractability. Yet the existence of this tradeoff does not mean that progress on developing methods that can deal effectively with *at least some* of the interesting problems that can be represented in an expressive modeling language is not possible and necessary. This is indeed the main concern of our work.

There are several types of planning models, emphasizing different aspects of reality that might be of interest. We will be concerned with the simplest of such models, the *classical planning* model, where actions are assumed to be deterministic, knowledge is assumed to be perfect, and a single agent is assumed to be acting in the world. We focus on mechanisms that are able to plan effectively with high-level expressive modeling languages for classical planning, and this will shed light on the relation between representation and computation, i.e. on the way in which an expressive language might be able to capture relevant structure of the problem that can be exploited to computational benefit, and which can otherwise become lost or concealed if the problem is represented using a lower-level language. For it seems to be the case that in a much-necessary quest for efficiency, research in planning has been shifting away from expressive but undecidable formalisms such as McCarthy's Situation Calculus towards lower-level but computationally convenient languages such as the STRIPS family. On the way there, we have perhaps become too accustomed to languages where the representation of even the simplest of concepts (a number,

a function) requires some involved workaround. It is not hard, however, at least for the author of this thesis, to recall the perplexity produced by the first encounter with these limitations. This perplexity provided no doubt some of the impetus that originally motivated the work on this thesis.

Our contribution in this thesis can be broadly split up in two parts. On the first, we build on a previously existing (but somewhat ignored) classical planning language, Functional STRIPS, a first-order formalism with support for function symbols. We show that not only function symbols allow more compact encodings, as was previously known, but that they can also allow more effective computations. To do so, we take existing heuristics based on the relaxed planning graph and extend them to deal with this language, and we show how they capture certain constraints that can render them more informative than their standard counterparts. In an analog development, we take yet another first-order feature, existential quantification, whose presence in standard modeling languages had been so far somewhat neglected, and make a similar case. We extend our heuristics to support existential quantification and show that this is advantageous on at least two accounts. From the modeling standpoint, formulas with existential variables are the natural way of modeling many common situations, including, for instance, situations where some choices can be left open for the solver to make them. Existential quantification additionally exposes an intimate connection between planning and the field of constraint satisfaction. From the computational standpoint, we empirically show that our heuristics can be more performant than previous approaches, because of their use of constraint satisfaction techniques specifically geared towards dealing with existential quantification.

On the second part of the thesis, we present a novel algorithm that builds on recently-developed notions of width and novelty in planning to perform an extremely effective exploration of the state space that *completely ignores the declarative representation of actions*, i.e. uses them as black boxes. The fact that such an algorithm is able to match state-of-the-art planning performance with such a big handicap, we argue, is a striking departure from classical planning research in the last decades. This handicap is however not a frivolous choice, but instead perfectly fits with the leitmotiv of the thesis: if we can plan effectively without looking at the representation of the actions, then we can define those actions with the mechanisms that best suit the problem at hand, including for instance expressive declarative modeling features, or non-declarative (i.e. procedural) definitions for actions with complex dynamics. Our algorithm hence advances the cause of expressive modeling languages in planning and the scope and effectiveness of classical planners.

Most of the results discussed in this thesis are the product of the work carried out by its author during the last four years, and have been previously published in a number of conference articles (Francès and Geffner, 2015; Ferrer-Mestres et al., 2015; Francès and Geffner, 2016a,b; Francès et al., 2017). The full reference for every article follows, together with the chapters where the results from that article are discussed:

- G. Francès and H. Geffner. Modeling and computation in planning: Better heuristics from more expressive languages. In *Proceedings of the 25th International Conference on Automated Planning and Scheduling*, pages 70-78, 2015. [Chapters 3 and 6]
- J. Ferrer-Mestres, G. Francès, and H. Geffner. Planning with state constraints and its application to combined task and motion planning. In *PlanRob, Work-*

shop on Planning and Robotics, 25th International Conference on Automated Planning and Scheduling, pages 13-22, 2015. [Chapters 3 and 6]

- G. Francès and H. Geffner. Effective planning with more expressive languages. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 4155–4159, 2016. [Chapters 3 and 6]

- G. Francès and H. Geffner. E-STRIPS: Existential quantification in planning and constraint satisfaction. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence*, pages 3082–3088, 2016. [Chapters 4 and 6]

- G. Francès, M. Ramírez, N. Lipovetzky and H. Geffner. Purely Declarative Action Representations are Overrated: Classical Planning with Simulators. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 4294–4301, 2017. [Chapters 5 and 6]

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> From this, one can make a deduction which is quite certainly the ultimate truth of jigsaw puzzles: despite appearances, puzzling is not a solitary game: every move the puzzler makes, the puzzlemaker has made before; every piece the puzzler picks up, and picks up again, and studies and strokes, every combination he tries, and tries a second time, every blunder and every insight, each hope and each discouragement have all been designed, calculated, and decided by the other.
>
> Georges Perec, *Life A User's Manual*

1.1 Planning and Artificial Intelligence

The nature of knowledge and of human intelligence has been one of the central concerns of Western philosophy at least since Descartes' Discourse on the Method (Descartes, 1996; Appiah, 2003), and a foundational cornerstone for the entire field of Artificial Intelligence (AI) since the seminal texts by Alan Turing (1950) and Claude Shannon (1950). A key component of human intelligence is *planning*, which in its broadest sense can be defined as the ability to look into the future in order to determine the most appropriate behavior for the present (Seligman et al., 2016). Planning lies at the heart of the Artificial Intelligence enterprise since its beginnings (McCarthy et al., 2006; Newell and Simon, 1963), and is critically related to the assumption that a *symbolic representation* or *model* of the world is a necessary condition for intelligence (McCarthy and Hayes, 1969; Newell and Simon, 1976). In light of this assumption, which is far from being universally accepted,[1] but which we share, planning can be more precisely defined as the *model-based approach* to intelligent behavior, where predictions based on a symbolic model of the world and objectives of the agent are used to figure out what to do next.

Different computational models of planning have been studied over the years, taking into account real-world features such as the existence and interaction of multiple

[1] See e.g. (Dreyfus, 1979; Brooks, 1990) for some interesting critiques to this symbolic approach to AI, or Nilsson (2007) and Dreyfus (2007) for a recent retrospective on the issue.

intelligent agents, the imperfection of human knowledge or the apparently stochastic nature of human actions (Russell and Norvig, 2009). This work is concerned with the simplest of those models, the so-called *classical planning* model, in which a single, omniscient agent is presumed to be acting in a world where actions have well-known deterministic effects, and her objective is to achieve through her action a certain state of affairs. The classical planning problem is no doubt an extremely simplistic model of the world, but it is broad enough to capture many real-world combinatorial problems, and a source of insight into how more complex models can be dealt with computationally. Indeed, one of the crucial known facts about such a simple model is that it is *intractable* (assuming a compact representation of it), and yet humans plan on an everyday basis with apparent ease. Another such fact is that there usually are many alternative ways to represent *the same* classical planning problem, and not all of them are equivalent in computational terms. The relation between representation and computation is indeed one of the main focuses of this work.

Progress and Challenges in Planning

In the classical planning problem, a compact description of the initial situation of the world and of the possible actions to be taken, along with a similar description of the goal to be achieved, are provided as input. A solution to the problem is a sequence of actions that maps the initial situation into some situation that satisfies the goal description; such a sequence is called a *plan*. A crucial distinction in planning is between finding *any* plan that solves a problem (*satisficing* planning) and finding a plan with minimum cost or number of actions (*optimal* planning). The focus of this work is on satisficing planning. The exact form of each of the above components (world state, action and goal descriptions) is deliberately left unspecified, as it will depend on the concrete representation language used to encode the problem. Some archetypal classical planning problems include the Blocks-world and well-known games such as the 8-puzzle, Sokoban or the Tower of Hanoi. In recent years there has been an increased effort to model tasks from other fields such as computer security or biology as (not necessarily classical) planning problems (Haslum, 2011; Gefen and Brafman, 2011; Hoffmann, 2015; Matloob and Soutchanski, 2016).

One of the main challenges of planning is at the same time one of its its most conspicuous strengths: *generality*. The ability to solve any problem represented in a *high-level declarative language* is attractive not only because it may shed light on the way human intelligence works, but also because modeling problems in such a language is much easier than programming an ad-hoc solver for every problem that might arise (McCarthy, 1987; Geffner, 2002). Many of the planning problems which are routinely used as benchmarks for novel algorithms are not difficult problems in themselves, particularly when optimal plans are not a requirement (Helmert, 2003, 2006b), and effective strategies can be easily found by humans for some of them. The challenge of planning is indeed to solve them *when represented in a general language*, where no domain-dependent information or control advice is available (Junghanns and Schaeffer, 1999). Solving problems encoded in the standard representation languages is in the worst-case PSPACE-complete (Bylander, 1994; Bäckström and Nebel, 1995), but despite this theoretical hardness, the field has witnessed significant practical progress in the recent decades. One of the causes of this has been methodological: the standardization of a modeling language (PDDL) and creation of a readily available set of benchmarks, along with the establishment of the International Planning

Competition (McDermott, 2000), has helped focusing efforts and having effective means of testing and comparing the performance of different planning algorithms and systems. Such a focus has its downsides too, as it might prevent the recognition and exploration of ideas that progress off the beaten track, in particular in what concerns representational issues (Rintanen, 2015), which is one of the focuses of this work. In any case, undeniable progress has been made in the scalability of planners and in the stock of ideas powering these planners, which nowadays includes approaches based e.g. on heuristic search, symbolic search or propositional satisfiability, among others (Kautz and Selman, 1992; Blum and Furst, 1995; Bonet and Geffner, 2001b; Hoffmann and Nebel, 2001a; Edelkamp and Reffel, 1999; Richter and Westphal, 2010).

This progress, however, does not mean that there is not still much to be done. In spite of frequent claims to the contrary, the scalability of planners on provably easy problems such as the abovementioned blocks-world is still far from adequate. Fundamental aspects of human cognition such as *learning* from previous solving attempts or from problem analogies, or coming up with *generalized* solutions which can be applied to several instances of the same problem, are likely to play a crucial role in effective planning, but still need to be explored more systematically (Martín and Geffner, 2004; Hu and De Giacomo, 2011; Geffner, 2010). On the representational side, the lack of expressiveness of standard planning languages is still a major issue, which acts both as an impediment for a wider adoption of the technology and sometimes as a computational obstacle (Rintanen, 2015). This thesis addresses some of these issues.

Modeling and Computation

There is a fundamental distinction between the precise mathematical *models* that characterize different classes of problems and the *languages* which are used to *represent* or *encode* problems that fall within a given class. This is not at all exclusive of planning, but holds in several areas of artificial intelligence such as SAT, constraint satisfaction, Answer Set Programming or Bayesian Networks (Biere et al., 2009; Dechter, 2003; Brewka et al., 2011; Pearl, 1988). In general, solvers tackle a precise class of problems, and modeling languages provide a way to represent the particular values taken by the different mathematical structures of a model in any instance of that model, hopefully in a general and succinct manner. But just as human language and thought are inextricably related (Gleitman and Papafragou, 2005; Boroditsky, 2011), so do their computational counterparts. Formal modeling languages do much more than representing problems in a declarative manner: they inevitably emphasize some aspects of the problem and conceal others, and by doing so they might ease or harden the task of solving the problem. This is usually referred to as preserving or concealing the *structure* of the problem. If planning is in general intractable, exploiting the structure of planning problems is likely to be key for solving efficiently those problems *that can be solved efficiently*, unless we are content with exponential search strategies. The impact of the language which we use to represent a problem on the efficiency with which the problem can be dealt with has long been recognized as a fundamental issue, both at the human level (Novick and Bassok, 2005) and at the computational level (Amarel, 1968). Several areas of AI have dealt with this issue (Freuder, 1999; Walsh, 2000; Smith, 2006), including planning (Bäckström, 1994, 1995; Nebel, 2000; Riddle et al., 2011; Barták and Vodrážka, 2015).

These considerations are related to what Doyle and Patil (1991) have named the "restricted language thesis", implicit in two influential articles by Levesque and Brachman (1985, 1987). Levesque and Brachman convincingly argue that, in the context of knowledge representations, there is a tradeoff between how expressive a representation language is and how tractable it is to reason over that language. Full first-order logic, for instance, is widely regarded as having an adequate degree of expressiveness, but logic entailment is, in general, not decidable. On the other hand, a knowledge base in database form (i.e. a collection of function-free atoms) is, under a few standard assumptions, much more convenient from a computational point of view, but its expressiveness is much more limited. Different formalisms, according to Levesque and Brachman, occupy different positions on the tradeoff between expressiveness and tractability. The restricted language thesis is the idea that we should focus our attention on restrictions or *fragments* of the knowledge representations where reasoning is tractable (i.e. polynomial), since otherwise there is little hope in developing systems which show some degree of intelligence.

According to Doyle and Patil, however, the very idea of sacrificing expressiveness for the sake of tractability is counterproductive, as it "destroys the generality of the language":

> These restrictions severely impair the utility of the system for developing applications. Language restrictions impose large costs in working around restrictions, when this is possible. When this is not possible, users must invent mechanisms external to the representation system for expressing their knowledge. In addition to reintroducing intractability concerns, this lack of standards for expression results in different ad hoc extensions to the language for each application. These gaps in expressive power thus mean that the restricted languages are not general purpose, contrary to the intent of their designers.
>
> (Doyle and Patil, 1991, p.5)

Tom Bylander extends the argument to the particular case of planning, and indeed shows that the syntactic restrictions on the STRIPS planning language that would be necessary (according to his particular taxonomy) for planning to be tractable are so severe that the resulting language can barely capture any interesting problem. Such restrictions, additionally, "would fail to permit expression of crucial domain knowledge" (Bylander, 1994). Indeed, a number of researchers have tried to characterize tractable fragments of planning along different syntactical or structural restrictions, but the resulting classes of problems are often considered to be too narrow to include problems that are interesting in practice (Bäckström and Klein, 1991; Jonsson and Bäckström, 1998; Giménez and Jonsson, 2008; Katz and Domshlak, 2008; Chen and Giménez, 2010). Recently, other researchers have alternatively argued in favor of more expressive languages, on both modeling and computational grounds (Ivankovic and Haslum, 2015; Rintanen, 2011, 2015). This is also the line that we take in this thesis. We will argue and show with concrete examples that seeking efficiency by restricting ourselves to low-level languages might actually be counterproductive. For the tradeoff between expressiveness and tractability is a worst-case result, but fixed a problem, it might well be that a high-level representation exposes its structure in a clearer manner, and thus lends itself to be tackled more efficiently by a solver.

1.2 Contributions

Our work in this thesis aims at developing effective means for dealing with expressive planning languages which go beyond the propositional fragment of STRIPS that is the *de facto* standard in the community. We think this is important not only because it eases the task of modeling, but also, as argued above, because higher-level modeling languages better expose the relevant bits of the problem structure that can be exploited computationally for efficiency purposes.

These are the main results of this thesis:

1. We build on the existing Functional STRIPS modeling language (Geffner, 2000), and show that the additional expressiveness obtained by allowing function symbols in the language can be exploited computationally to derive more accurate heuristics. We do this by extending the *relaxed planning graph* construct to support these function symbols, and showing that this permits taking into account some constraints in the problem that would otherwise become lost in function-free representations. This is illustrated with a number of examples and experiments.
2. Similarly, we argue that existentially quantified variables are an essential modeling feature. They allow to elegantly and explicitly model *open choices*, i.e. situations in which there is no need to commit to a particular variable instantiation, and they are key for representing and reasoning with constraints in planning. We further argue that the standard strategy of compiling them away is not a good option, as it hides relevant problem structure that can be exploited computationally. We show this by extending the relaxed planning graph to account for existential quantification through constraint satisfaction techniques, and proving the resulting heuristics more informed. We demonstrate the importance of existential variables with a set of novel planning domains, which additionally illustrate an important bridge between constraint satisfaction and planning problems. We report experiments that show that the performance of our approach clearly dominates that of previous approaches.
3. Building on the recent notions of planning width and novelty of a state (Lipovetzky and Geffner, 2012), we develop a family of *simulation-based* planning algorithms which perform an extremely effective search which does not require a compact declarative representation of the actions of the problem. This represents a significant departure from mainstream planning research over the last decades. Despite the severe handicap that having no declarative information about actions represents, experimental results over a large set of standard benchmarks from the last two international planning competitions show that the performance of the approach is competitive with the state of the art. The prospect of an efficient planning technique that (mostly) does away with the requirement of declarative representations is extremely interesting in itself, and additionally furthers our agenda. As it turns out, if no information about action structure is necessary, then expressive modeling features can be used to define them. Even more interestingly, these features do not need to be declarative, but one can instead use procedural routines to provide the denotation of parts of the language.
4. On a more practical side of things, an important contribution of this thesis is the FS planner, which to the best of our knowledge is the first planner that supports

the full specification of the Functional STRIPS language, including arbitrary terms using (possibly nested) function symbols and externally-defined symbols, plus a number of interesting extensions, such as efficient support for existential quantification, state constraints (Lin and Reiter, 1994; Ferrer-Mestres et al., 2015), and global constraints (Régin, 2011). Most of the problems we use to benchmark the performance of all of the above ideas have been made public and are also novel, and therefore should be regarded as a contribution. Some of them have already been used by other authors (Scala et al., 2016b).

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 introduces most of the necessary background on classical planning, organized along the axes of model, language and computation, as well as a review of the state of the art in satisficing planning. Additional background knowledge on first-order logic can be found in Appendix A. Chapters 3 to 6 are relatively self-contained. Chapter 3 discusses and evaluates our extension of traditional heuristics based on the relaxed planning graph to deal with function symbols in the modeling language. Likewise, Chapter 4 discusses and evaluates the extension of the same heuristic to support existential quantification. Chapter 5 presents our width-based algorithms that are able to plan with no knowledge on the action structure, along with an experimental study with benchmarks from the last two international competitions that shows these algorithms are competitive with the state of the art. Chapter 6 describes the `FS` planner where all of the ideas presented in the previous chapters have been implemented. We conclude the thesis in Chapter 7 by summarizing our findings and outlining some lines of ongoing or future research.

Chapter 2

Background

In this chapter we review the area of planning, with particular emphasis on classical planning, articulating our discussion along three main axes: models, languages and computation. We also briefly describe the basic ideas from satisfiability and constraint satisfaction, which are related to our work.

2.1 Introduction

One of the oldest and most central areas in artificial intelligence, planning can be best defined as the *model-based approach* to the task of generating autonomous behavior automatically (Geffner and Bonet, 2013). In planning, unlike in other close disciplines such as reinforcement learning, a precise model of how the world works is used to derive the behavior of an agent. This model might typically include a description of which of the myriad features of the world are relevant for the task at hand, the possible effects of actions on those features, a description of the current situation and of the desired outcome of the agent's actions. Computing such a behavior is usually worst-case intractable, and hence a great deal of the research carried out in the area is focused on the ways in which this can be done *in practice*.

Crucial to planning, as well as to other AI fields such as satisfiability or constraint satisfaction, is the distinction between

1. the *mathematical models* that characterize in a precise manner the problem that we are trying to solve,
2. the *formal languages* that are used to represent the problems within one of those models, and
3. the different *computational approaches* that can be used to solve those problems,

on which we will elaborate later on this chapter. Several planning models have been considered in the literature, with different degrees of *expressiveness*, which might capture or ignore things such as the presence of other planning agents in the environment, the unpredictable nature of the agents' actions, the non-atomic nature of those actions, or the lack of knowledge the agent can have about the environment. The more expressive a model is, in general, the more difficult that reasoning and planning for it can be expected to be (Levesque and Brachman, 1985). Indeed, a

salient strand in artificial intelligence and knowledge representation research concerns the balance between the competing needs of providing expressive modeling languages and of reasoning about problems expressed in those languages in an effective manner.

2.2 The Classical Planning Model

The work presented in this thesis is chiefly concerned with the most elementary of those models, *classical planning*, in which an omniscient agent is assumed to act alone in a world where no changes occur that are not triggered by the agent's actions, and the outcome of these actions is fully-deterministic. Despite the simplicity of such a model, work in classical planning is relevant not only because it provides a starting point which is broad enough to capture many real-world combinatorial problems, but also because it often provides insight into how more expressive models can be dealt with computationally (Yoon et al., 2007).

We next give a formalization of this classical planning model as a transition system, and will later use this formalization to provide in a clear and compact manner the semantics of several of the languages that we will be discussing.

Definition 2.1 (Classical Planning Model)**.** *A* classical planning model *is a tuple* $\Pi = \langle S, s_0, S_G, O, f \rangle$ *that consists of*

- *A finite set of states* S*,*
- *An initial state* $s_0 \in S$*,*
- *A set of goal states* $S_G \subseteq S$*,*
- *A set of operators* O*, and*
- *A (partial) transition function* $f : S \times O \mapsto S$*.*

The transition function encodes the result $s' = f(o, s)$ of applying an operator o in state s. To emphasize that not all operators are applicable in all states, it is customary to denote by $A(s) \subseteq O$ the set of operators applicable in state s:

$$A(s) = \left\{ o \in O \mid \exists s' \; f(s, o) = s' \right\}$$

It is often necessary to include a *cost function* $c : O \times S \mapsto \mathbb{R}^+$ that captures the cost of applying a certain action in a given state. Although some of the results that we present in this thesis can be easily extended to this *classical planning with costs* setting, we will in general assume that we are dealing with uniform cost schemas, in which the cost of all actions is the same, which is in general equivalent to saying that all actions have unit cost. For simplicity, we will thus often ignore this cost function.

Definition 2.2 (Plan)**.** *Given a classical planning model* Π*, a* plan *is a sequence of operators* $\pi = \langle a_0, \ldots, a_n \rangle$ *that* maps *the initial state* s_0 *into a goal state* $s \in S_G$*, i.e.* π *induces a sequence of states* $s_0, \ldots, s_{n+1}$*, where* $s_{n+1} \in S_G$ *and for all* $i = 0, \ldots, n$ *we have that* $s_{i+1} = f(a_i, s_i)$*.*

Any classical planning model $\Pi = \langle S, s_0, S_G, O, f \rangle$ can be readily mapped into a directed graph G_Π where nodes represent planning states, edges are labeled with

operators from O, and a directed edge (v_1, v_2) between two nodes with label $o \in O$ represents the existence of an operator o whose application transforms the state represented by v_1 into the state represented by v_2. The problem of finding a plan reduces then to the problem of finding a path between the graph vertex representing s_0 and any other vertex representing a state in S_G. The computational cost of such a task is polynomial in the number of vertices; in planning, however, one is usually interested in finding a path in a graph whose size is exponential with respect to the *compact representation* that is used to describe it implicitly. The task of a *classical planner* is precisely to take a compact representation of a classical planning problem and to produce a plan for it. *Satisficing planning* is concerned with computing a plan, regardless of its cost or length, whereas *optimal planning* is additionally concerned with the computed plan being the shortest or lowest-cost of all possible plans.

Compact representations are usually expressed in a declarative planning language such as STRIPS or PDDL. Planning languages provide not only a way to represent problems over completely different domains in a uniform manner, but also to reveal problem structure that can be exploited computationally. Most techniques which are key to the performance of modern planners, heuristic or otherwise, are derived from the actual representation of the problem, *not from the characteristics of the underlying planning model alone.*

2.3 Classical Planning Languages

In this section we briefly review some of the planning languages that have been used over the years, paying particular attention to STRIPS and PDDL, because of their widespread use nowadays, and to Functional STRIPS, which will constitute the basis of our work. There is, to the best of our knowledge, no updated, systematic comparison of the different planning languages commonly used in planning, likely because the community has leaned over the years towards using the standard PDDL. Bäckström (1995) and Nebel (2000), however, present interesting theoretical analyses of the comparative expressive power of different modeling formalisms.

2.3.1 The Situation Calculus

The Situation Calculus (McCarthy, 1963; McCarthy and Hayes, 1969; Reiter, 2001), developed after John McCarthy's *Advice Taker* (McCarthy, 1960), is one of the earliest attempts to have an expressive first-order mechanism to reason about actions and change. The Situation Calculus promotes a full axiomatization in first-order logic of the effects of actions; in the calculus, the states of the world, named *situations*, and the relevant actions of the problem, are represented as objects of a particular type. Any predicate or function symbol is *extended* to account for an extra situation argument, so that e.g. an atom $on(a, b, s)$ might denote that a block a is on top of a block b *in a certain situation* s. The result of applying an action a in a certain state s is denoted by $result(a, s)$, where $result$ is a reserved function symbol. One of the effects of applying the action $a \equiv move(b_1, b_2)$ in the classical blocks-world, for instance, might be denoted with the axiom

$$\forall s \left[b_1 \neq b_2 \wedge clear(b_1, s) \wedge clear(b_2, s) \rightarrow on(b_1, b_2, result(a, s)) \right]$$

The expressiveness of the formalism however comes at a cost. In the Situation Calculus, planning is indeed reduced to (first-order logic) theorem proving (Green, 1969), which in the general case is not decidable (Enderton, 2001; Rautenberg, 2006). Variants and extensions of the Situation Calculus, however, have been used in recent years for expressing dynamics and control (Levesque et al., 1997; De Giacomo et al., 2000).

2.3.2 STRIPS

Partly as a response to the computational challenge posed by the Situation Calculus, the Stanford Research Institute Problem Solver (STRIPS, Fikes and Nilsson (1971)) brought about not only an actual problem solver, but also a novel modeling language, less expressive than the Situation Calculus, but more suited for efficient computation techniques. In the original STRIPS formulation, *world models* are collections of first-order sentences over a given language $\mathcal{L}$. These sentences can be simple atoms such as $at(rob_1, room_2)$, or sentences such as

$$\forall x, y, z \; connects(x, y, z) \rightarrow connects(x, z, y)$$

A STRIPS *operator* o is a tuple $o = \langle name(o), pre(o), add(o), del(o) \rangle$, where $name(o)$ is a symbol identifying the operator, the *precondition* $pre(o)$ is a sentence over $\mathcal{L}$, and the *add* and *delete* lists $add(o)$ and $del(o)$ are sets of such sentences over $\mathcal{L}$. A STRIPS system $\Sigma = \langle M_0, O \rangle$ is made up of an initial world model M_0 plus a set of operators O. A *plan* π of a given system Σ is a sequence of operators $\pi = (o_1, \ldots, o_n)$, which in turn induces a sequence of world models $M_0, \ldots, M_n$ such that M_0 is the initial world model and

$$M_i = (M_{i-1} \setminus del(o_i)) \cup add(o_i)$$

A plan π is valid if $M_{i-1} \vdash pre(o_i)$ for all i.

From a historical point of view, one of the key technical contributions of STRIPS is its solution to the *frame problem* (McCarthy and Hayes, 1969), embodied in the assumption that those aspects of the world not explicitly mentioned in the operators' add and delete list remain unaffected (Fikes and Nilsson, 1993). Lifschitz (1987) however notices that the semantics of this original formulation of the language is ill-defined, because of the infinite nature of the sets of first-order sentences. Lifschitz proposes a reformulation where world models, add and delete lists can only be *sets of atoms* instead of sentences. For the system to be proven *sound*, Lifschitz observes, the only non-atomic sentences allowed in M_0 and in the add list of any operator should be sentences which are satisfied in all possible states of the world. The standard practice since is to follow this restriction.

The original specification of the language additionally included several features such as action expansions or safety constraints (Weld and Etzioni, 1994). For reasons of succinctness and relevance to our work, however,we limit our description here to the STRIPS subset of PDDL (McDermott et al., 1998), which we formalize next.

In terms of the language syntax, STRIPS allows only *constant* and *predicate* symbols. The only terms that the language permits are constant symbols such as b_1, b_2, and atoms are recursively defined in the usual way by (a) applying a predicate symbol p of arity k to k constants $t_1, \ldots, t_k$ to form the atom $p(t_1, \ldots, t_k)$, and (b) applying the standard logical connectives $\neg$, $\wedge$, $\vee$.

Definition 2.3 (STRIPS Planning Problem)**.** *A STRIPS planning problem* $P = \langle A, O, I, G \rangle$ *consists of*

- *A set of atoms* A*,*
- *A set of operators* O*,*
- *A set of atoms* $I \subset A$ *describing the initial state, and*
- *A set of atoms* $G \subset A$ *describing the goal state.*

Often, planning literature makes a distinction between problem *domains* and *instances*, in the sense that many planning problems can be seen as different instances of the same domain, perhaps with different properties, initial or goal situations, etc. We will not be concerned with such a distinction here.

Operators have the form that was described above. The set of operators is often given through compact *first-order action schemas* containing variables, which are assumed to be existentially quantified. We will have more to say about this later, but in general, unless explicitly noted, we will assume that operators are grounded.

The precise semantics of any STRIPS problem P, and in general of a classical planning problem expressed in any language, can be given by a mapping with a corresponding classical model $\mathcal{S}(P)$. In this case, $\mathcal{S}(P) = \langle S, s_0, S_G, O', f \rangle$ is defined as follows:

- The set of states is $S = \mathcal{P}(A)$,
- The initial state is $s_0 = I$,
- The set of goal states is given by $S_G = \{s \in S \mid G \subseteq s\}$,
- The set of operators is $O' = O$
- The transition function f is defined only over those pairs $\langle s, o \rangle$ such that $pre(o) \subseteq s$, in which case

$$f(s, o) = (s \setminus \mathrm{del}(o)) \cup \mathrm{add}(o)$$

Note that there is a fundamental difference between the semantics of the sets of atoms I and G. The set I is intended to denote *a single state* in S, namely, the state where the atoms in the set I are true, and the rest are assumed to be false.[1] In contrast, the set G is intended to denote *a subset of states* in S, made up of all those states for which the atoms in G are true, regardless of whether there are other true atoms or not.

In STRIPS, actions modify the denotation of some of the predicate symbols of the language, which are then called *fluents*. The denotation of other symbols not affected by any action, as well as that of constant symbols, is assumed not to change, for which reason they are called *fixed* symbols.

[1] This assumption is often referred to as the *closed-world* assumption (Reiter, 1978).

2.3.3 Action Languages and ADL

The eighties and nineties saw the development of languages based on first-order logic that tried to reach an intermediate ground between the expressiveness of Situation Calculus and the computational convenience of STRIPS (Pednault, 1989; Baral et al., 1997; Gelfond and Lifschitz, 1998). Pednault's Action Description Language (ADL; Pednault, 1986, 1989, 1994) is perhaps the most important of these. ADL can be seen as a set of restrictions on the form of the axioms allowed by the Situation Calculus that allow for the definition of STRIPS-like action schemas with function symbols, conditional effects and universal quantification.

Syntactically, each action schema is made up of a name, a set of variables (the action parameters) plus the *precondition*, *add*, *delete* and *update* lists. The precondition is a first-order formula, usually a conjunction, that determines applicability of the action; the add and delete lists are sets with one of the forms

1. $R(t_1, \ldots, t_n)$ if ϕ.
2. $R(t_1, \ldots, t_n)$ for all $z_1, \ldots, z_m$ such that ϕ.

where R is a relation symbol, all t_i's are terms, all z_i's are variables and ϕ a (first-order) formula. Finally, the update list is the equivalent of the add and delete lists for function symbols, and has one of the forms

1. $f(t_1, \ldots, t_n) \leftarrow t$ if ϕ.
2. $f(t_1, \ldots, t_n) \leftarrow t$ for all $z_1, \ldots, z_m$ such that ϕ.

where f is a function symbol of arity n (possibly a constant, i.e. $n = 0$), t all t_i's are terms, all z_i's are variables and ϕ a (first-order) formula.

Semantically, ADL states are the actual algebraic structures that underlie first-order logic interpretations (Rautenberg, 2006).[2] ADL schemas define not first-order formulas that hold or cease to hold in the state that results from applying the schema, as in the original STRIPS, but instead they directly define the transformation on the resulting state of the world, i.e. on the first-order interpretation representing this state. ADL allows for partially-known initial initial states, going beyond the classical planning model; when the initial state is fully specified, however, the problem can be dealt with techniques which are significantly simpler than the more generic regression mechanism proposed in (Pednault, 1994).

A few planners have been developed for fragments of the ADL language (McDermott, 1991; Penberthy and Weld, 1992). To the best of our knowledge, none of them supports the full language specification and, in particular, none of them supports function symbols. ADL is nonetheless interesting to us because it introduces many of the features that make Functional STRIPS the convenient language upon which the work described in Chapters 3 and 4 is based. Functional STRIPS is formally introduced and discussed in Section 2.3.6.

In parallel, several expressive *action languages* have been also developed (Gelfond and Lifschitz, 1993, 1998), including the action language $\mathcal{A}$, which is roughly equivalent to the propositional fragment of ADL; and action languages $\mathcal{B}$ and $\mathcal{C}$. $\mathcal{B}$ extends $\mathcal{A}$ with *static laws*, i.e. axioms for inferring the truth value of certain fluents; $\mathcal{C}$ extends

[2] This idea is further elaborated in the description of the Functional STRIPS language below.

$\mathcal{B}$ with *non-inertial fluents*, i.e. fluents whose value might change *not* as the result of some action enacted by the agent (Gelfond and Lifschitz, 1998).

2.3.4 SAS$^+$

The *Extended Simplified Action Structures* SAS$^+$ formalism (Bäckström and Nebel, 1995), deriving from SAS (Bäckström and Klein, 1991), is a slight generalization of propositional STRIPS which allows for multivalued state variables. It can be seen that, under the *ESP reduction* framework presented by Bäckström (1995), both STRIPS and SAS$^+$ are *equivalent* from an expressive point of view, in the sense that a problem represented in any of the two formalisms can be translated in polynomial time into the other formalism, and solution sizes are preserved. This would seem to suggest that there is no practical difference between using e.g. SAS$^+$ or STRIPS to represent a problem that needs to be solved. This, however, is not at all the case, since polynomial reducibility is too coarse a measure, and nothing prevents the problem represented in one language to be exponentially harder to solve than the (polynomially-transformed) equivalent in the other language (Bäckström, 1994). As a matter of fact, this can actually happen when transforming certain multivalued SAS$^+$ problems into propositional STRIPS (Bäckström, 1994).

2.3.5 PDDL

The Planning Domain Definition Language (PDDL; McDermott et al., 1998) was developed as a standardization effort for the first edition of the International Planning Competition in 1998 (AIPS-98, then called the AI planning systems competition; McDermott, 2000), and has been since then used in all subsequent editions. PDDL should therefore be seen more as a *standard syntax* for the core of the STRIPS language, along with certain extensions coming from ADL and other planner-specific existing modeling languages, than as a distinct language in itself. The original PDDL formulation contemplated the use of features such as conditional effects, universal quantification over dynamic universes, stratified axioms or safety constraints, not all of which have resisted equally well the test of time. The success of PDDL prompted however the development of several extensions to accommodate more sophisticated modeling needs. PDDL 2.1, for instance, offered support for numeric fluents and of non-atomic actions, i.e. actions with variable duration (Fox and Long, 2003), and sparked an interesting controversy over the nature of modeling languages and the relation between expressiveness and computability (Bacchus, 2003; Boddy, 2003; Geffner, 2003; McDermott, 2003a; Smith, 2003). PDDL 3.0 (Gerevini et al., 2009) and PDDL 3.1 further extended the language with support for trajectory constraints and preferences. PDDL 3.1, in particular, addressed previous criticism for the lack of support for function symbols (McDermott, 2003a) by extending the language with so-called *object fluents*, essentially a reformulation of the Functional STRIPS language that we present below.[3] Unfortunately, no planner entering subsequent international planning competitions seems to have offered support for it, and no benchmark in the competition makes use of it either. Other separate extensions exist as well, such as PDDL+, which deals with continuous change and exogenous processes (Fox and Long, 2002; McDermott, 2003b).

[3] See http://icaps-conference.org/ipc2008/deterministic/PddlResources.html [Accessed 28 Jun. 2017].

```
;; Problem Domain
(define (domain hanoi)
  (:requirements :strips)

  (:predicates (clear ?x) (on ?x ?y) (larger ?x ?y))

  (:action move :parameters (?disk ?from ?to)
    :precondition (and (larger ?to ?disk) (on ?disk ?from)
                       (clear ?disk) (clear ?to) (not (= ?from ?to)))
    :effect (and (clear ?from) (on ?disk ?to)
                 (not (on ?disk ?from)) (not (clear ?to))))
)

;; Problem Instance
(define (problem hanoi3) (:domain hanoi)

  (:objects peg1 peg2 peg3 d1 d2 d3)

  (:init
    (larger peg1 d1) (larger peg1 d2) (larger peg1 d3)
    (larger peg2 d1) (larger peg2 d2) (larger peg2 d3)
    (larger peg3 d1) (larger peg3 d2) (larger peg3 d3)
    (larger d2 d1) (larger d3 d1) (larger d3 d2)

    (clear peg2) (clear peg3) (clear d1)

    (on d3 peg1) (on d2 d3) (on d1 d2))

  (:goal (and (on d3 peg3) (on d2 d3) (on d1 d2))))
```

Figure 2.1: PDDL encoding of a towers of Hanoi problem with 3 towers.

For the sake of brevity, we do not provide here a full formalization of the language and semantics of the several variations of PDDL.[4] The basic STRIPS subset of the language introduced in Section 2.3.2 will cover most of our needs; those extensions relevant to our work such as functional symbols are properly formalized in Section 2.3.6 below. We conclude this section by presenting and discussing a sample encoding of the well-known Tower of Hanoi.

Example 2.4 (Tower of Hanoi PDDL encoding)**.** *Figure 2.1 shows a possible PDDL representation of the classic Tower of Hanoi problem with* 3 *disks. Constants (that is, PDDL objects) include three pegs and three disks. A predicate* larger *encodes size relations between disks and also between disks and pegs (note that all pegs are larger than all disks, so that disks can always be placed on empty pegs). As is customary, in the initial situation all disks are stacked on the first peg, respecting their sizes, and the goal is to move them so that they lay in the same order on the third peg. The encoding of the* move *action takes care that stacking size constraints are respected, and that only* clear *disks are moved.*

[4] Indeed, the modular design of PDDL makes it unclear that the semantics of the language is well defined for certain combinations of the several extensions of the language (McDermott, 2003b).

2.3.6 Functional STRIPS

Functional STRIPS (FSTRIPS, for short) is a classical planning language that extends STRIPS with functional symbols in order to provide a number of expressive and computational advantages, such as the ability of making indirect reference to domain objects through the use of nested terms. Although the original formulation by Geffner (2000) presents a variable-free version of the language, we here lift that restriction to give a full first-order logic account, which we will need in subsequent chapters. We assume the reader is familiar with the basic definitions of first-order logic, on which the following FSTRIPS formalization relies; an overview of these definitions can be found in Appendix A. We will use the classical blocks-world as a running example.

Problem Language. A Functional STRIPS problem P is defined over a many-sorted first order logic language with equality, which we denote by $\mathcal{L}(P)$. Such a language is made up of a non-empty and finite set T of *types* (or *sorts*), a possibly infinite set of variables $v_1^t, v_2^t, \ldots$ for each type $t \in T$, a set Φ of *function symbols* and a set Π of *relation symbols*, assumed to include the equality symbol "=". A Functional STRIPS language $\mathcal{L}(P)$ meant to model blocks-world, for instance, will typically include a unary predicate symbol *clear*, a binary function symbol *loc*, and constant symbols b_1, b_2, b_3 and *table*.

Problem Interpretations. Recall that any first-order interpretation $\mathcal{M}$ is made up of a set of typed, non-empty universes $\mathcal{U}_\mathcal{M} = \{\mathcal{U}_t\}_{t \in T}$ plus the denotation of each function and predicate symbol, where:

1. The denotation of a constant (nullary function) symbol c of type $\langle t \rangle$ is an element $c^\mathcal{M} \in \mathcal{U}_t$.
2. The denotation of a nullary predicate symbol P is a truth value $P^\mathcal{M} \in \{\top, \bot\}$.
3. The denotation of an n-ary function symbol f of type $\langle t_1, \ldots, t_n, t_{n+1} \rangle$ is a function $f^\mathcal{M} : \mathcal{U}_{t_1} \times \cdots \times \mathcal{U}_{t_n} \to \mathcal{U}_{t_{n+1}}$.
4. The denotation of an n-ary predicate symbol P of type $\langle t_1, \ldots, t_n \rangle$ is a subset $P^\mathcal{M} \subseteq \mathcal{U}_{t_1} \times \cdots \times \mathcal{U}_{t_n}$.

Functional STRIPS assumes that the universe of discourse in any problem P is fixed, i.e. all valid interpretations for P have the same universe, denoted by $\mathcal{U}_P = \{\mathcal{U}_t\}_{t \in T}$. It is further assumed that *the universe $\mathcal{U}_t$ of any type t is finite*. In the blocks-world example above, a possible interpretation $\mathcal{M}$ for $\mathcal{L}(P)$ might consist of a single type t with universe $\mathcal{U}_t = \{b_1, b_2, b_3, table\}$. Constants b_1, b_2, b_3 and *table* are respectively denoted by the object with equal name; the denotation of predicate symbol *clear* is the set $clear^\mathcal{M} = \{table, b_1\}$; the denotation of function symbol *loc* is the function:

$$loc^\mathcal{M}(x) = \begin{cases} b_2, & \text{if } x = b_1 \\ b_3, & \text{if } x = b_2 \\ table, & \text{if } x = b_3. \end{cases}$$

Action Schemas. A Functional STRIPS *action schema* a is defined by its *name*, its *signature* $\langle v_1, \ldots, v_n \rangle$, where each v_i is a (typed) variable, its *precondition formula* $\text{pre}(a)$, which is an arbitrary formula over $\mathcal{L}(P)$, and its *set of (conditional) effects* $\text{effs}(a)$. Each effect $e \in \text{effs}(a)$ can be either a relational or a functional effect:

- A *relational effect* e has the form $\phi \to L$, where ϕ is a formula (the *condition* of the effect) and L is a literal, i.e. an atomic formula $R(\bar{t})$ or its negation $\neg R(\bar{t})$. In the first case we say that the effect is an *add effect*, in the second, a *delete effect*.
- A *functional effect* e has the form $\phi \to f(\bar{t}) := w$, where ϕ is again the condition of the effect, w is an arbitrary term, and $f(\bar{t})$ is made up of a (possibly nullary) function symbol f over a tuple of terms $\bar{t}$ of appropriate size and type. The types of term w and symbol f must coincide.

In our blocks-world running example, the only action schema is *move*, with signature $\langle x, y \rangle$, precondition

$$x \neq table \wedge x \neq y \wedge loc(x) \neq y \wedge clear(x) \wedge clear(y)$$

and effects

1. $\top \to loc(x) := y$,
2. $\top \to clear(loc(x))$, and
3. $y \neq table \to \neg clear(y)$.

As customary, when the condition of an effect is $\top$, we often drop it and denote e.g. the first effect above simply as $loc(x) := y$.

Fluent and Fixed Symbols. A function symbol f is said to be *fluent* if it appears in the head of some functional effect $\phi \to f(\bar{t}) := w$, whereas a relation symbol R is fluent if it appears in the head of some relational effect $\phi \to R(\bar{t})$ or $\phi \to \neg R(\bar{t})$. Non-fluent function and relation symbols are called *fixed*, or *static*. In the blocks-world above, *clear* and *loc* are the only fluent symbols.

Ground Actions. Action schemas can be grounded by substituting all occurrences of the action parameters (i.e. variables in the signature of the schema) in the action precondition and effects by type-consistent *constant symbols* (we disallow the possibility of grounding action schemas with arbitrary terms, as in ADL (Pednault, 1994)). We denote by $\mathrm{gr}(a)$ the *grounding* of an action schema a with signature $\langle x_1, \ldots, x_n \rangle$. $\mathrm{gr}(a)$ is a set of action schemas with nullary signature (which we will call *operators*, or simply *ground actions*), and contains one ground action for every possible substitution for variables $x_1, \ldots, x_n$. We further denote by $\mathrm{gr}(A)$ the grounding of a *set of action schemas*, $\mathrm{gr}(A) = \cup_{a \in A} \mathrm{gr}(a)$.

States and State Variables. States in a FSTRIPS problem P *are* the representation of first-order interpretations over $\mathcal{L}(P)$ with fixed universe $\mathcal{U}_P$.[5] In practice, states need only represent the denotation of fluent symbols, since the denotation of fixed symbols remains by definition the same for all possible states of the problem. We often write c^* to refer to the actual (state-independent) denotation of such fixed symbols. To simplify definitions, we will also assume that every fixed constant symbol c denotes a different object c^*, and that for every type t and element $o \in \mathcal{U}_t$,

[5] This is similar to the way ADL states are defined.

there is a fixed constant symbol c whose denotation c^* is o, i.e. all objects in the universe are denoted by some fixed constant.

The relation between fluent symbols and states is captured by the notion of *state variable*. For any fluent n-ary function symbol f with type $\langle t_1, \ldots, t_n, t_{n+1} \rangle$, and tuple $\langle a_1, \ldots, a_n \rangle \in \mathcal{U}_{t_1} \times \cdots \times \mathcal{U}_{t_n}$, $f(a_1, \ldots, a_n)$ is a (functional) state variable of type t_{n+1}. Relational state variables are defined analogously for relation symbols.

Because all universes in $\mathcal{U}_P$ are finite, the number of possible first-order interpretations with universe $\mathcal{U}_P$ is also finite, which implies that planning in FSTRIPS is decidable (and polynomial in the number of states). For the same reason, there is a finite number of state variables. We will often denote by $\mathcal{V}(P)$ the set of all state variables of a FSTRIPS problem P. Indeed, FSTRIPS states can be equivalently seen as first-order logical interpretations or as complete assignments of values to the set of state variables of the problem, mapping each functional state variable with type t to an element from $\mathcal{U}_t$, and each relational state variable to a truth value in $\{\top, \bot\}$.

We will sometimes use the notation $\mathcal{M}(s)$ to emphasize that we are referring to *the interpretation encoded by a state* s. We write t^s and ϕ^s for the denotation of term t and formula ϕ in state s. With a slight abuse of notation, we also say that the term $f(c_1, \ldots, c_m)$ is a state variable when $c_1, \ldots, c_m$ are *fixed constants* of the language, since in all valid interpretations for the problem, they actually correspond to the same state variable $f(c_1^*, \ldots, c_m^*)$. The same notational device will be used for predicate symbols. To avoid confusion between the term $f(c_1, \ldots, c_m)$ and the corresponding state variable, we will sometimes denote the latter by $\underline{f(c_1, \ldots, c_m)}$.

We are finally ready to define a FSTRIPS problem:

Definition 2.5 (FSTRIPS planning problem)**.** *A* FSTRIPS planning problem *is a tuple* $P = \langle \mathcal{L}(P), \mathcal{U}_P, s_I, A, \phi_G \rangle$*, where*

- $\mathcal{L}(P)$ *is a first-order language, as described above.*
- $\mathcal{U}_P = \{\mathcal{U}_t\}_{t \in T}$ *is the set of finite, typed universes.*
- s_I *is the initial state of the world.*
- A *is a set of action schemas.*
- *The* goal formula ϕ_G *is a formula over the* $\mathcal{L}(P)$ *language.*

The semantics of P are defined by giving its correspondence with a classical planning model $\mathcal{S}(P)$. The basic intuition is that FSTRIPS operators represent ways of modifying the denotation of some of the logical symbols in the interpretation that describes the state of the world. The transition function of the planning model induced by a FSTRIPS problem can be defined as follows:

Definition 2.6 (FSTRIPS Transition Function)**.** *The partial transition function* $f : S \times O \mapsto S$ *corresponding to a* FSTRIPS *problem* P *is defined over all state-operator pairs* $\langle s, o \rangle$ *such that* $\mathcal{M}(s) \vDash pre(o)$*. For one such pair, the state* $s' = f(s, o)$ *that results from applying operator* o *in state* s *is the logical interpretation with universe* $\mathcal{U}_P$ *where the denotation of each symbol in* $\mathcal{L}(P)$ *is as follows. For each*

function symbol f,

$$f^{\mathcal{M}(s')}(\bar{x}) = \begin{cases} w^{\mathcal{M}(s)}, & \text{if } o \text{ has a functional effect } \phi \to f(\bar{t}) := w \\ & \text{such that } \mathcal{M}(s) \vDash \phi \text{ and } \bar{x} = \bar{t}^{\mathcal{M}(s)} \\ f^{\mathcal{M}(s)}(\bar{x}), & \text{otherwise.}[6] \end{cases}$$

Analogously, the denotation $P^{\mathcal{M}(s')}$ of every n-ary predicate symbol P in $\mathcal{M}(s')$, for $n > 1$, contains the tuple $\bar{x}$ if both of the following conditions hold:

1. *There is no delete effect $\phi \to \neg P(\bar{t})$ such that $\mathcal{M}(s) \vDash \phi$ and $\bar{t}^{\mathcal{M}(s)} = \bar{x}$, and*
2. *either $\bar{x} \in P^{\mathcal{M}(s)}$ or there is an add effect $\phi \to P(\bar{t})$ such that $\mathcal{M}(s) \vDash \phi$ and $\bar{t}^{\mathcal{M}(s)} = \bar{x}$.*

The above can be extended to nullary predicate symbols P in a straight-forward manner. This generalizes the STRIPS transition function $f(s, o) = (s \setminus \mathrm{del}(o)) \cup \mathrm{add}(o)$ given above to the first-order case.

Let us now summarize the semantics of a Functional STRIPS problem:

Definition 2.7 (FSTRIPS semantics)**.** *Let $P = \langle \mathcal{L}(P), \mathcal{U}_P, s_I, A, \phi_G \rangle$ be a Functional STRIPS problem. The semantics of P are given by the classical planning model $\mathcal{S}(P) = \langle S, s_0, S_G, O, f \rangle$, where:*

- *The finite set of states S, as described above, contains all possible interpretations over the language $\mathcal{L}(P)$ with universe $\mathcal{U}_P$.*
- *The initial state is $s_0 = s_I$,*
- *The set of goal states S_G contains all interpretations $\mathcal{M}$ over $\mathcal{L}(P)$ such that $\mathcal{M} \vDash \phi_G$,*
- *The set of operators is $O = \mathrm{gr}(A)$,*
- *The transition function $f : S \times O \mapsto S$ is the function given in Definition 2.6.*

The denotation of fixed function and predicate symbols in FSTRIPS can be provided either extensionally, i.e. by enumeration in the initial state, or intensionally, through *external procedures* specified in some programming language. Additionally, for simplicity we will often assume that the denotation of standard fixed symbols such as "3" or "+" is given by the underlying programming language.

Example 2.8 (Blocks-world, complete)**.** *Let us fully recapitulate the blocks-world example with 3 blocks. The logical language of the problem P has one single type t, and symbols $\{clear, loc, b_1, b_2, b_3, table\}$, where clear is a unary predicate symbol, loc a binary function symbol, and the rest of symbols are constants. The fixed universe*

[6] If operator o has more than one functional effect $\phi_1 \to f(\bar{t}_1) := w_1$ and $\phi_2 \to f(\bar{t}_2) := w_2$ such that (1) both effect conditions are satisfied in $\mathcal{M}(s)$ and (2) $t_2^{\mathcal{M}(s)} = t_1^{\mathcal{M}(s)}$, but their right-hand sides w_1 and w_2 have different denotations $w_1^{\mathcal{M}(s)} \neq w_2^{\mathcal{M}(s)}$, then we consider the FSTRIPS problem inconsistent. In contrast with STRIPS, in FSTRIPS it is not possible to ensure that a given problem is consistent from the syntactical description of the problem alone.

$\mathcal{U}_t$ contains the objects $\{b_1, b_2, b_3, table\}$. The only action schema is move(x, y)*, with precondition*

$$x \neq table \wedge x \neq y \wedge loc(x) \neq y \wedge clear(x) \wedge clear(y)$$

and effects (1) $loc(x) := y$, (2) $clear(loc(x))$, and (3) $y \neq table \rightarrow \neg clear(y)$. A possible ground action in the set $\mathrm{gr}(A)$ *of the problem is $move(b_1, table)$. This ground action has precondition*

$$loc(b_1) \neq table \wedge clear(b_1) \wedge clear(table),$$

and effects (1) $loc(b_1) := table$ and (2) $clear(loc(b_1))$. Note that the substitution of action parameters by actual constant symbols often allows to simplify precondition and effects by following the standard rules of first-order logic.

Both clear and loc are fluent symbols, i.e. their denotation will possibly change in different states, because they appear in the head of the move *action. In contrast, the denotation of symbols b_1, b_2, b_3 and table is fixed in all states of the problem.*

A situation where block b_1 is on top of b_2, block b_2 is on b_3, and b_3 is on the table, is represented by the state (i.e. interpretation) s where the symbol clear is denoted by the set $clear^s = \{b_1, table\}$, and the symbol loc is denoted by the function

$$loc^s(x) = \begin{cases} b_2, & \text{if } x = b_1 \\ b_3, & \text{if } x = b_2 \\ table, & \text{if } x = b_3 \end{cases}$$

The value assigned to the point table by the function is actually irrelevant for this particular problem.

The problem has state variables $clear(table)$, $clear(b_1)$, $clear(b_2)$, $clear(b_3)$, $loc(b_1)$, $loc(b_2)$ and $loc(b_3)$. Any FSTRIPS state can be seen as an assignment of values to these state variables; the above state, for instance, can be seen as the assignment $clear(table) = \top$, $clear(b_1) = \top$, $clear(b_2) = \bot$, $clear(b_3) = \bot$, $loc(b_1) = b_2$, $loc(b_2) = b_3$, $loc(b_3) = table$. Finally, a typical goal formula for the problem could be $loc(b_1) = table \wedge loc(b_3) = b_2$.

Example 2.9 (n-puzzle)**.** *The blocks-world example above still does not show an actual computer-parsable problem specification. As was mentioned before, PDDL 3.1 does actually provide support for object fluents, a somewhat obscure equivalent mechanism to function symbols. We can thus use PDDL 3.1 to encode FSTRIPS problems; Fig. 2.2 shows an actual PDDL 3.1 encoding of the classical n-puzzle domain, along with a possible instance with $n = 8$.*

The problem types are tile *and* position*, with fixed universes $\mathcal{U}_{tile} = \{no\_tile, t_1, \ldots, t_8\}$ and $\mathcal{U}_{position} = \{p_1, \ldots, p_9\}$. The logical constants no_tile, $t_1, \ldots, t_8$, $p_1, \ldots, p_9$ have fixed denotations, as they are not affected by the* swap *action. The predicate symbol adjacent also has fixed denotation, which is provided extensionally in the specification of the initial problem state. In contrast, the logical constant blank, with type $\langle position \rangle$, has fluent denotation, as has the unary function tile_at, with type $\langle position, tile \rangle$.*

```
;; Problem Domain
(define (domain n-puzzle)
  (:types tile position)

  (:constants no_tile - tile)

  (:predicates (adjacent ?p1 ?p2 - position) )

  (:functions
    (tile_at ?p - position) - tile
    (blank) - position)

  (:action swap :parameters (?p - position)
  :precondition (adjacent ?p (blank))
  :effect (and
    (assign (blank) ?p)
    (assign (tile_at (blank)) (tile_at ?p))
    (assign (tile_at ?p) no_tile)))
)

;; Problem Instance
(define (problem example) (:domain n-puzzle)
  (:objects t1 t2 t3 t4 t5 t6 t7 t8 - tile
            p1 p2 p3 p4 p5 p6 p7 p8 p9 - position)

  (:init
    (adjacent p1 p2) (adjacent p2 p1)
    (adjacent p1 p4) (adjacent p4 p1)
    ...
    (adjacent p8 p9) (adjacent p9 p8)

    (= (blank) p5)
    (= (tile_at p1) t7) (= (tile_at p2) t2) (= (tile_at p3) t4)
    (= (tile_at p4) t5) (= (tile_at p5) no_tile) (= (tile_at p6) t6)
    (= (tile_at p7) t8) (= (tile_at p8) t3) (= (tile_at p9) t1))

  (:goal (and
                        (= (tile_at p2) t1) (= (tile_at p3) t2)
    (= (tile_at p4) t3) (= (tile_at p5) t4) (= (tile_at p6) t5)
    (= (tile_at p7) t6) (= (tile_at p8) t7) (= (tile_at p9) t8)))
)
```

Figure 2.2: Functional STRIPS encoding of the n-puzzle problem, for $n = 8$. Figure 2.3 illustrates the corresponding initial and goal states.

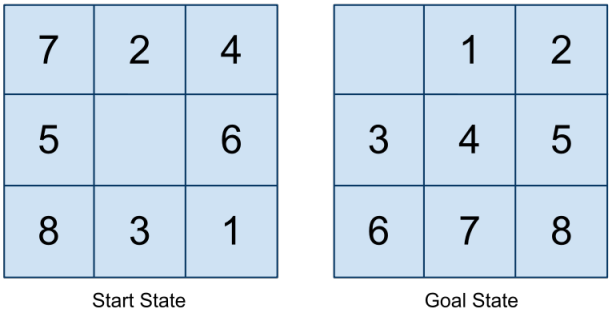

Figure 2.3: Graphical depiction of the initial and goal states of the 8-puzzle problem corresponding to the FSTRIPS encoding of Fig. 2.2. By Haiqiao [Creative Commons CC0 License], via Wikimedia Commons.

The initial and goal situations that correspond to the given encoding are depicted in Fig. 2.3. The 9 *positions are numbered left-to-right, top-bottom, from* p_1 *to* p_9*. The blank constant is intended to represent which of the* 9 *positions* p_i *is blank; for that position, it will always hold that* $tile\_at(p_i) = no\_tile$*. The action* $\text{swap}(p : position)$ *simply puts the tile on position* p *where the blank was.*

Correspondence Between Functional STRIPS and PDDL. Let us briefly clarify some possible confusion between the logical language we employ and the standard usage in PDDL problems. *PDDL objects* are logical constants with fixed denotation. *Nullary PDDL functions* are also logical constants, but might have fluent or fixed denotation, depending on whether they are affected by some action or not. The slightly misleading notion of *PDDL constant* is *not* related to logical constants, but rather refers to *PDDL objects* which invariably exist in all problems of a certain PDDL domain. In the n-puzzle example above, all PDDL objects, including the constant *no_tile*, are logical constants with fixed denotations; the fact that a PDDL object is declared as a PDDL constant or not pertains to the distinction between PDDL domain and PDDL instance, but is otherwise irrelevant to the logical formulation of the problem. *blank* is also a logical constant (i.e. nullary function symbol), but in this case with fluent denotation.

2.4 Computation

After presenting the classical planning model and some standard representational languages, we now discuss the main computational question in planning: given that planning is intractable, what are the most appropriate ways of computing plans? We first sketch the main tractability results, then survey some of the computational paradigms that have been proposed over the years, moving progressively towards those which are directly related to our work.

2.4.1 The Complexity of Planning

The two elementary planning-related problems from a complexity perspective are:

1. PLAN-EXISTENCE(P): Given a classical planning problem P, is there a plan that solves the problem?

2. PLAN-LENGTH(P, k): Given a classical planning problem P and a non-negative integer k, does a plan with at most k actions exist?

Both problems mirror the distinction between satisficing and optimal planning, and their computational complexity depends on the exact modeling language that we choose to represent the planning problem. Chapman (1987); Bylander (1994); Bäckström and Nebel (1995); Erol et al. (1995) are some of the standard references in this regard, and provide complexity results for problems encoded in STRIPS, SAS^+, and other formalisms; the following discussion is based on them. In the case of problems expressed in propositional STRIPS, PLAN-EXISTENCE is PSPACE-complete (Bylander, 1994), even when highly restricted versions of the problem are considered. It remains PSPACE-complete, for instance, when operators are allowed to have no more than one precondition, and likewise when they can have only one effect. The above assumes that the input of the computational problem is grounded; if instead action schemas are used, the problem complexity moves one step up the ladder to EXPSPACE-completeness, as the grounding can result in an exponential number of actions (Ghallab et al., 2004).

Two results with strong practical implications that provide the basis for some of the most successful heuristic techniques in planning (discussed below) are the following. First, PLAN-EXISTENCE is NP-complete for *delete-free* STRIPS, i.e. STRIPS where actions have no delete effects. This is because in that setting, the number of atoms satisfied in each state grows monotonically with the application of actions, and thus the length of any plan is (polynomially) bounded by the total number of ground atoms, making the problem lie within NP. Interestingly, what makes delete-free PLAN-EXISTENCE complete for the class NP is the interaction between add effects and negative preconditions: if negative preconditions are also disallowed, then PLAN-EXISTENCE is polynomial (Bylander, 1994; Erol et al., 1995). This is of crucial importance for the so-called delete-free relaxation of any planning problem.

Allowing function symbols in the language somewhat muddies the waters. Chapman (1987) directly claims that "efficient general purpose planning with more expressive action representations is impossible". The model that Chapman has in mind is general enough to encode any Turing machine as a planning problem, and includes the possibility of creating novel objects (i.e. logical constants) through the application of actions. This of course makes the state space of the search infinite and places the problem well beyond what we have defined as the *classical* planning model. Erol et al. (1991, 1992, 1995) and (Subrahmanian and Zaniolo, 1995, Sec. 2) discuss several other undecidability results for planning with function symbols. In the case of Functional STRIPS as defined above (Section 2.3.6), however, the universe of discourse in any problem is *finite* and remains the same in all logical interpretations relevant for the problem. Since the number of symbols in the language is also finite and fixed, the number of possible interpretations will be finite, and hence checking the problem of satisfiability of any given logical sentence is decidable, as one can simply enumerate all possible interpretations $\mathcal{M}$ and evaluate the truth value of the sentence under $\mathcal{M}$. We will see however that more efficient strategies exist for that.

Between Theory and Practice

The worst-case theoretical intractability of the classical planning problem might sometimes seem at odds with the good observed empirical performance of actual

planners when applied to some of the benchmarks which are standard in the community. This can always of course be attributed to these particular benchmarks possessing some kind of structure that allows current planning techniques to deal with them effectively, but not too much is known from a theoretical point of view in that respect.

A starting point to address this issue is to understand what is the actual complexity of the planning benchmarks when viewed as computational problems on their own, i.e. when tackled not through a generic planner in some planning language, but through ad-hoc algorithms. A number of researchers, for instance, discuss the complexity the popular blocks-world domain along with several variations of it (Chenoweth, 1991; Gupta and Nau, 1992; Slaney and Thiébaux, 2001). Gupta and Nau (1992) establish that (the nowadays most common variation of) blocks-world optimal planning is NP-hard, while blocks-world satisficing planning is in P. More interestingly than the particular classification is the fact that the authors are able to pin down the *structural* reason of the theoretical hardness of optimal planning, which in this case is related to the positive side-effects of actions. Helmert (2003, 2006b) extends the above thread by providing analyses of the complexity of both satisficing and optimal planning for several benchmarks from the International Planning Competitions, which interestingly ranges from P to PSPACE. A similar analysis, this time of the approximation properties of several competition benchmarks, is presented in (Helmert et al., 2006). Helmert and Röger (2008), in turn, discuss the scalability limits of optimal-planning search algorithms such as A* when almost perfect heuristics, i.e. where heuristic error is bound by an additive constant, are used. Their findings show that in many standard planning problems, even when the heuristic error is small, an exponential number of node expansions might be necessary when using such algorithms.

A more theoretical take on the above problem has focused on attempting to characterize tractable subclasses of planning, defined either by syntactical or structural properties of the problems (Bäckström and Klein, 1991; Jonsson and Bäckström, 1998; Giménez and Jonsson, 2008; Chen and Giménez, 2010), although these subclasses are often considered to be too narrow to include problems that are interesting in practice. More recently, alternative analysis tools such as fixed-parameter complexity analysis are starting to be used in an attempt to obtain finer-grained parametrizations of the complexity of solving the classical planning problem (Backstrom and Jonsson, 2011; Bäckström et al., 2012; Kronegger et al., 2013; Bäckström et al., 2015).

2.4.2 Computational Paradigms in Planning

From the practical point of view, the problem of automated planning has been approached in two significantly distinct manners. Beginning with the *General Problem Solver* (GPS; Newell et al., 1959), planning has been thought of as a problem-solving activity best tackled through state-space search algorithms that *find* a sequence of actions in some suitably-designed search space. At the same time, and beginning with McCarthy's *Situation Calculus* (McCarthy, 1963; Levesque et al., 1998; Reiter, 2001), planning has been conceptualized as a theorem-proving activity, where predicate calculus is used on a rigorous axiomatization of the effects of actions on the world to *deduce* a sequence of actions. In this thesis we focus mainly on the first approach.

At the same time, the short history of planning is punctuated by certain shifts in the type of *computational approaches* deemed as best suited to approach the problem (Hendler et al., 1990; McDermott and Hendler, 1995; Weld, 1999; Minker, 2000; Ghallab et al., 2004; Geffner and Bonet, 2013). The General Problem Solver, for instance, computed plans through *means-ends analysis*, a form of search that relied on an analysis of the *differences* between the current and the desired state of the world — differences which depend on the operators that are available to modify that state of the world. Strongly inspired by the way human cognition tackles problem-solving (Newell and Simon, 1963), the GPS was a first attempt at abstracting away the particular details of each problem to derive general principles.

The research carried out at the Stanford Research Institute during the late 1960s and early 1970s resulted in several developments with lasting impact on the field of AI.[7] Besides the STRIPS modeling language (described in Section 2.3.2), the project brought about the development of the Stanford Research Institute Problem Solver (Fikes and Nilsson, 1971), a GPS-like planner in charge of controlling the Shakey robot. The planner anticipated delete-free heuristics in that it ignored the delete lists of operators to focus on operators that added logical clauses which would help progress a search for a state that entailed the goal formula (Fikes and Nilsson, 1993). Up until the mid-seventies, most planning systems usually proceeded by performing some kind of search in state-space (Hendler et al., 1990).

The paradigm shifted with the advent of *partial-order planning* (alternatively called *least commitment planning* or *nonlinear planning*), where plans are searched not in state (world) space but in a space where search nodes represent partially-constructed plans (Sacerdoti, 1975; Tate, 1977; McAllester and Rosenblitt, 1991; Weld, 1994). Starting with an empty plan, plans are refined by adding actions and ordering constraints between them, until the set of actions and constraints in a search node can be used to derive an actual plan. One of the advantages of least commitment planners is that they avoid unnecessary commitments to concrete (total) action orderings, sidestepping for instance the problems of state-space search with symmetries. A good representative of partial-order planning was UCPOP (Penberthy and Weld, 1992), which supported a fragment of the ADL language and was considered state-of-the-art during the first half of the 1990s.

The second half of the 1990's, however, witnessed the arrival of several novel planning techniques. On the one hand, Blum and Furst (1995) developed GRAPHPLAN, which builds a polynomial *planning graph* of the problem from the initial state, and then performs backward search from the goal using information extracted from the graph to guide the search; if no valid plan is found, the planning graph is further extended and the process repeated, in a strategy that resembles iterative-deepening search (Weld, 1999). On the other hand, Kautz and Selman showed that determining whether a classical planning task has a plan of a certain *horizon* length can be converted into a propositional formula in conjunctive normal form that is then fed into a SAT solver, in what became known as the *planning as satisfiability* paradigm (Kautz and Selman, 1992, 1996; Kautz, 2006).

At the same time, state-space planners regained the attention of the community

[7] Which included, among others, the development of the A* algorithm (Hart et al., 1968) and of the first automatic generation systems of both planning macros (Fikes et al., 1972) and abstraction hierarchies (Sacerdoti, 1974). Fikes and Nilsson (1993) offer a brief but interesting retrospective analysis.

thanks to the development of automatic methods to generate domain-independent heuristics from the problem description (McDermott, 1996; Bonet et al., 1997; Bonet and Geffner, 2001b), quickly sparking the development of additional improvements to the general idea (Hoffmann and Nebel, 2001a). The performance of *heuristic search planners*, particularly in satisficing (i.e. non-optimal) planning, quickly exceeded that of previous approaches, becoming to dominate the first editions of the International Planning Competition. Most of the work in this thesis can indeed be framed within the *planning as heuristic search* paradigm, whose main ideas we survey in the next section.

Other paradigms have been explored as well, and although their empirical performance has not always matched that of contemporary state-of-the-art techniques, they represent interesting contributions and sometimes perform well for particular problem classes and/or for non-classical planning models. These include *planning as CSP* (Do and Kambhampati, 2001), *planning as integer programming* (Kautz and Walser, 1999; Vossen et al., 1999) or *planning as model checking* (Edelkamp and Reffel, 1999; Edelkamp and Helmert, 2001; Torralba et al., 2017), as well as recent hybridizations of the above techniques such as the one presented in (Davies et al., 2015).

2.4.3 Planning as Search

The founding assumption that underlies the *planning as search* paradigm is that the classical planning problem can be cast as a path-finding problem over a directed graph unambiguously defined by the planning problem specification, in which the nodes of the graph represent the states of the planning problem, its (labeled) edges represent the transitions between states induced by the application of planning operators, and the task is to find a path between the node representing the initial state of the planning task and any node representing a goal state. Figure 2.4 presents a possible visualization of the state space of a blocks-world problem with three blocks. Clearly, any path-finding algorithm can be used to solve the classical planning problem; the main obstacle though is that the cost of standard path-finding algorithms such as Dijkstra's is polynomial in the size of the state space, and, as Fig. 2.4 suggests, the size of the state space induced by a planning task is exponential in the number of problem variables. Altogether, this means that standard algorithms will not scale up unless some additional mechanism is in place to improve the search.

One such common mechanism to improve path-finding algorithms is to use information about the goal nodes to *guide* the search. This is most often achieved by means of *heuristic functions*, which aim at providing quick estimates of the (minimum) cost $h(s)$ of reaching a goal state from any given state s (Pearl, 1983; Edelkamp and Schroedl, 2011). Algorithms making use of this technique are called *heuristic search* algorithms, as opposed to *blind search* (also known as *brute-force*) algorithms, in which the goal plays no role in providing search guidance. Common examples of blind search algorithms include breadth-first search, depth-first search, uniform cost search (Dijkstra's algorithm) or iterative-deepening depth-first search, all possessing distinct time and memory costs (Russell and Norvig, 2009). Examples of heuristic search algorithms, on the other hand, include greedy best-first search, hill-climbing, A* (Hart et al., 1968) or IDA* (Korf, 1985). Before describing them, however, we turn to the question of how can heuristic functions be obtained.

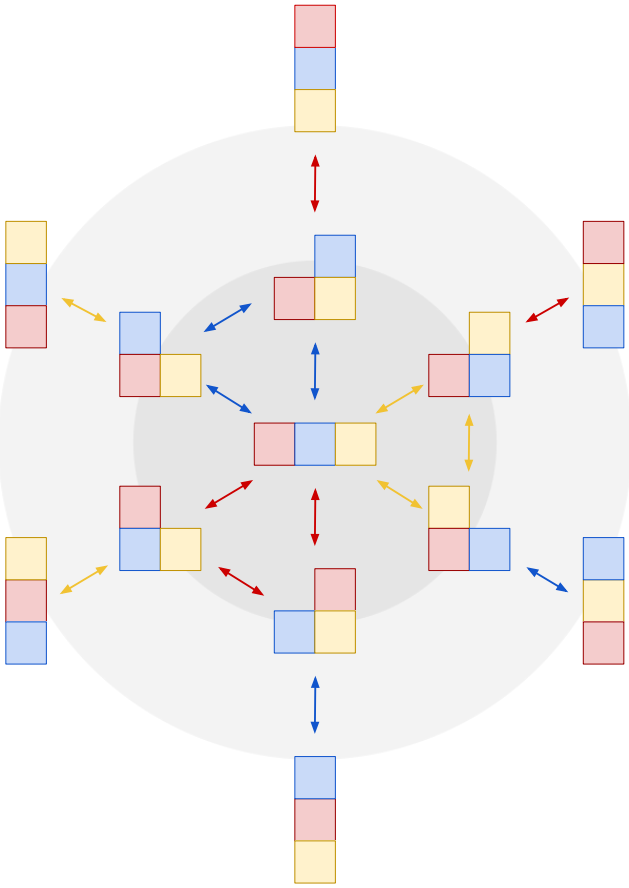

Figure 2.4: Complete state space of a blocks world with three blocks and an atomic move action. All edges are bidirectional, since any block move can can be immediately reversed, and they are colored with the color of the moved block.

Heuristics are often computed from *relaxations*, i.e. simplifications of the problem we want to solve, where certain restrictions have been lifted or reformulated in order to make the relaxed problem easier, yet at the same time informative enough so that a solution to this relaxed problem provides information that is relevant to solve the original problem. A classic example is the well-known 8-puzzle problem and the *Manhattan distance* heuristic. For any particular configuration s of an 8-puzzle board, a common heuristic choice is to set $h(s)$ to the sum of the Manhattan distances between each tile and its goal position; where the Manhattan distance between two positions of the board is defined as the sum of the absolute differences of their Cartesian coordinates. In this case, the heuristic can be seen as the length of an optimal solution for a relaxation of the original 8-puzzle, namely, the relaxation in which the constraints on tile moves are loosened and tiles can be moved to adjacent positions regardless of whether those positions are empty or not. Interestingly, it turns out that in the 8-puzzle this heuristic never overestimates the true cost to a goal, since no matter which optimal solution for the original problem we pick, it will necessarily feature *at least* n_k moves for each tile k, where n_k is the Manhattan distance between the position of the tile and its goal position (Russell and Norvig, 2009). Heuristics that never overestimate the true minimum cost are called *admissible*; admissibility is necessary e.g. to guarantee that solutions returned by A* are optimal.

2.4.4 Classical Planning Heuristics

As a matter of fact, the key idea behind the planning as search paradigm is indeed to derive heuristics in an automatic and domain-independent manner from relaxations of the problem that are defined *from the encoding of the problem* in some modeling language. The prime example of this is the *delete-free relaxation* (McDermott, 1996, 1999; Bonet et al., 1997; Bonet and Geffner, 2001b), which has arguably had a crucial impact on many of the developments in both satisficing and optimal heuristic-based planning in the last two decades (Haslum and Geffner, 2000; Porteous et al., 2001; Bonet and Geffner, 2001a; Hoffmann and Nebel, 2001a; Hoffmann, 2003; Vidal, 2004a; Haslum et al., 2005; Richter et al., 2008; Keyder and Geffner, 2008; Helmert and Domshlak, 2009; Keyder et al., 2012; Domshlak and Nazarenko, 2013; Domshlak et al., 2015). The delete-free relaxation of a STRIPS problem P, often denoted by P^+, is an abstraction of the original problem in which the delete effects of actions are ignored. While finding an optimal plan in a delete-free problem is still NP-hard (Bylander, 1994), two key properties of the relaxation, *monotonicity* and *decomposability*, ensure that that a (possibly suboptimal) plan can be computed in a tractable manner. Monotonicity refers to the fact that in a delete-free problem, the number of atoms satisfied in any reachable state grows monotonically with the application of new actions, since no atom is ever "deleted" from a state. This bounds plan length by ensuring that in any plan for P^+ every operator needs to appear no more than once, since its effects are never undone. Decomposability ensures that a plan for a conjunction $p \wedge q$ can be obtained simply by concatenating a plan for p and a plan for q.

Other options not based on the delete-free relaxation are also possible to obtain heuristics in an automatic, domain-independent manner. One such option is the *critical-path* family of heuristics h^{m} (Haslum and Geffner, 2000), which computes lower-bound estimates of the cost of achieving sets of atoms of size m, for some fixed integer $m > 1$. In this case, the relaxation resides in the assumption that the cost of reaching a large set of atoms is not larger than the cost of reaching its most expensive subset of size at most m. Another option is to abstract some of the characteristics of the problem away, for instance by ignoring certain atoms of the problem. In general, *abstraction* heuristics build on a homomorphism α that maps states of the original problem into states of an abstract state space, typically of smaller size. The heuristic value $h^{\alpha}(s)$ of any state s is then obtained as the minimum distance between $\alpha(s)$ and any goal state in the abstract state space. Different ways of constructing the homomorphism function lead to different heuristics of this class, which for instance include pattern database heuristics (Edelkamp, 2001; Haslum et al., 2005, 2007) and merge-and-shrink heuristics (Helmert et al., 2007).

A third option are (pseudo-) heuristics based on the notion of *landmarks* (Porteous et al., 2001; Hoffmann et al., 2004), facts that must necessarily hold at some point in any plan for the problem. Landmarks can be propositional formulas over the problem language (most commonly single atoms, but the use of conjunctions and disjunctions of atoms has also been explored), or over atoms $use(a)$ that denote the necessary use of some problem action a in any plan. The computation of landmarks is in general intractable (Porteous et al., 2001), but several tractable strategies exist to approximate useful sets of landmarks (Richter et al., 2008; Bonet and Helmert, 2010), a simple one of which is to check if the delete-free relaxation of the problem *without one single action* becomes unsolvable, in which case that action must necessarily be

part of any plan. The computation of landmarks is usually distinguished from their use. Landmarks can e.g. be used in an heuristic-like fashion by setting $h(s)$ to the number of landmarks that have not been achieved in the way to the state s; this is not a heuristic in the standard sense, since the value of $h(s)$ is path-dependent, but it works well in practice. Other, more sophisticated uses of landmarks exist (Helmert and Domshlak, 2009). Critical-path, abstraction-based and landmark heuristics have all been proven effective strategies to tackle classical planning problems.

Delete-Free Relaxation Heuristics

Because of their relevance to this thesis, we describe a bit more extensively some heuristics based on the delete-free relaxation.

Definition 2.10 (Delete-free relaxation of a STRIPS problem)**.** *Let P be a STRIPS problem $P = \langle A, O, I, G\rangle$. The* delete-free *relaxation of P, denoted by P^+, is the STRIPS problem $P^+ = \langle A, O^+, I, G\rangle$, where*

$$O^+ = \{\langle pre(o), add(o), \emptyset\rangle \mid o \in O\}$$

In words, the relaxation consists in ignoring the delete-effects of the actions. The optimal cost of a plan for a delete-free relaxed problem P^+ starting from any state s is usually denoted by $h^+(s)$. As we just saw, in general the computation of h^+ NP-hard (Bylander, 1994), so the computation of a fast heuristic does typically involve some further relaxation.[8] Three of the most prominent options in that sense give rise to the $h_{\max}$, h_{add} and h_{FF} heuristics. We briefly present the first two here, and devote the next subsection to the h_{FF} heuristic and the *relaxed planning graph* upon which it builds, directly related to our work in Chapters 3 and 4.

The $h_{\max}$ and h_{add} heuristics (Bonet and Geffner, 2001b) approximate the value of h^+ by making assumptions on *the cost $c(Q, s)$ of achieving, from a given state s, all atoms in a set Q* (think of Q as an action precondition or problem goal). $h_{\max}$ assumes that this cost is equal to the cost of achieving the most expensive atom in the set, while h_{add} assumes that this cost is equal to the sum of the costs of achieving each of the atoms in the set — both of them are "incorrect" assumptions, due to the fact that actions have side-effects, and an action achieving a certain atom $p \in Q$ can help achieve another atom $q \in Q$ as well, but they are tractable approximations nevertheless.

Definition 2.11 ($h_{\max}$ and h_{add} heuristics)**.** *Let $P = \langle A, O, I, G\rangle$ be a STRIPS problem, s a state in the corresponding state space, and $Q \subseteq A$ a set of atoms. The* max-cost $c_{\max}(Q, s)$ *of achieving Q from s is:*

[8] However, see (Betz and Helmert, 2009) for an interesting per-domain analysis of the actual complexity of computing h^+ for many of the standard classical planning benchmarks in a domain-dependent fashion and how well (or poorly) are they approximated in practice by the heuristics used by classical planners, or (Helmert and Mattmüller, 2008) for a similar analysis focused on admissible heuristics.

$$c_{\max}(Q,s) = \begin{cases} 0, & Q \subseteq s \\ \min_{o \in ach(q)} \Big(1 + c_{\max}(pre(o), s)\Big), & Q = \{q\} \\ \max_{q \in Q} \; c_{\max}(q,s), & \text{otherwise} \end{cases}$$

where for any atom $q \in A$, $ach(q) = \{o \in O \mid q \in add(o)\}$ is the set of operators that achieve q. The h_{max} value of any state s is then

$$h_{max}(s) = c_{\max}(G,s)$$

The add-cost $c_{\text{add}}(Q,s)$ *can be defined analogously by replacing the* max *function by the sum function:*

$$c_{\text{add}}(Q,s) = \begin{cases} 0, & Q \subseteq s \\ \min_{o \in ach(q)} \Big(1 + c_{\text{add}}(pre(o), s)\Big), & Q = \{q\} \\ \sum_{q \in Q} \; c_{\text{add}}(q,s), & \text{otherwise} \end{cases}$$

And then,

$$h_{add}(s) = c_{\text{add}}(G,s)$$

Both $h_{\max}$ and h_{add} can be efficiently computed in time polynomial in the number of fluents, e.g. through a Bellman-Ford-like procedure (Bonet and Geffner, 2001b). The $h_{\max}$ heuristic is admissible, although usually not too informative, and can in fact be seen as the special case for the h^{m} heuristic in which $m = 1$ (Haslum and Geffner, 2000). The h_{add} heuristic is *not* admissible, but is often in practice a much better choice for satisficing planning, as it is more informed.

The Relaxed Planning Graph

Another option is to approximate h^+ as the cost of a suboptimal *relaxed plan* (i.e. a plan for the problem relaxation), which can be computed by a `GRAPHPLAN`-like procedure (Blum and Furst, 1995) applied to the delete-free problem, through what is known as the *Relaxed Planning Graph* (RPG) (Hoffmann and Nebel, 2001a).

Definition 2.12 (Relaxed planning graph)**.** *The relaxed planning graph RPG(P, s) of a given STRIPS problem $P = \langle A, O, I, G\rangle$ built from state s is a succession $P_0, A_0, P_1, A_1, \ldots$ of interleaved* atom *(P_i) and* action *(A_i) layers, such that:*

- *Atom layer $P_0 = s$ contains all atoms which are true in the state s,*
- *action layer $A_i = \{o \in O \mid pre(o) \subseteq P_i\}$ contains all actions* applicable in the previous atom layer P_i*, and*
- *atom layer $P_{i+1} = P_i \cup \big(\bigcup_{a \in A_i} add(a)\big)$, for $i \geq 0$, contains all atoms from the previous atom layer P_i plus those atoms achievable through the actions in A_i.*

As the number of atoms is finite, this inductive definition must eventually reach a fixpoint k where $P_i = P_k$ for all $i > k$. The number of atom layers k is upper-bounded by the (finite) number of atoms, which guarantees that an iterative implementation

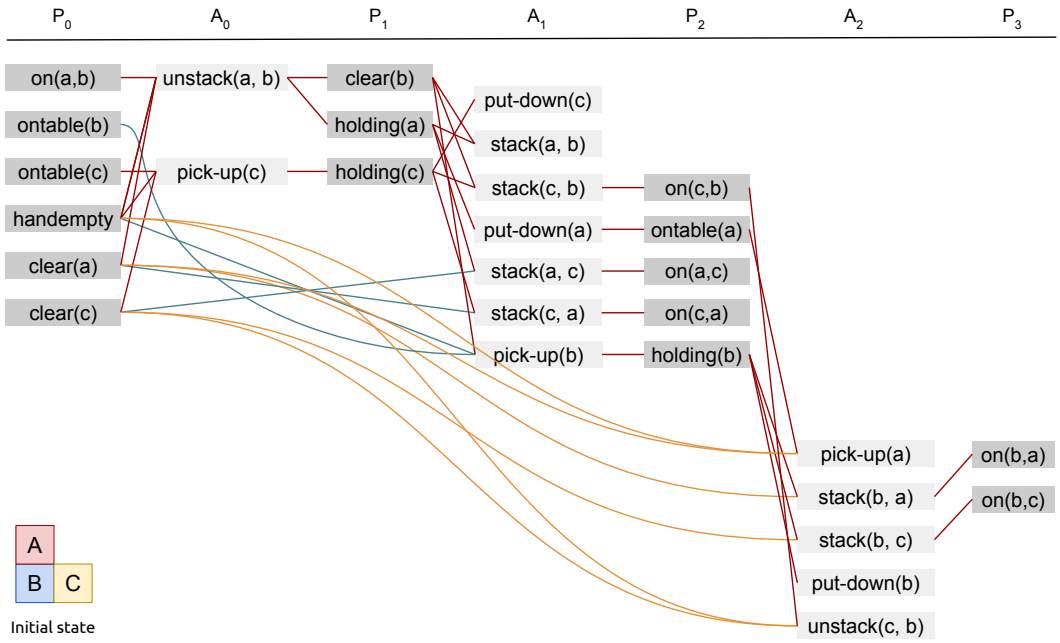

Figure 2.5: A (partial) Relaxed Planning Graph corresponding to a blocks-world instance with three blocks a, b, and c, and initial state as depicted. The illustration alternates atom (P_i) and action layers (A_i); lines denote support both between atom and actions and between actions and atoms. For readability, some actions have been removed, actions are only depicted on the first layer they become applicable, and atoms on the first layer where they become reachable.

of the above definition will eventually finish (in time polynomial with respect to the number of atoms). Figure 2.5 depicts the relaxed planning graph of a toy blocks-world instance with three blocks, assuming the standard 4-action STRIPS encoding of the problem with actions *stack, unstack, pick-up* and *put-down*. P_3 is a fixpoint and contains all problem atoms. In general, P_i is an *over-approximation of the set of atoms that are reachable in i steps*: if an atom is indeed reachable in i steps, it will be in P_i.

The relaxed planning graph from a state s can be used to *extract* a relaxed plan π_{RPG} *backwards* from the first atom layer A_k that reaches all goal atoms, i.e. such that $G \subseteq A_k$. Very roughly, the extraction algorithm selects for each goal atom $p \in G$ one of the actions $o \in A_{k-1}$ that achieves the atom; o is usually called a *supporter* of p, and different selection mechanisms are possible when more than one such supporter exists. Each selected supporter action o is tagged, and the algorithm proceeds recursively by seeking a support for each of the precondition atoms in $pre(o)$. The procedure takes polynomial time, and once it is finished, the relaxed plan π_{RPG} can be built by linearizing all tagged actions, respecting the order given by the first layer A_i in which they appear. Because multiple supporters for any atom might exist, the resulting relaxed plan, as well as heuristics based on it, are not unique. Figure 2.6 illustrates the extraction algorithm on the toy problem from Fig. 2.5. Bryce and Kambhampati (2007) offer a clear and complete exposition of different aspects of the RPG.

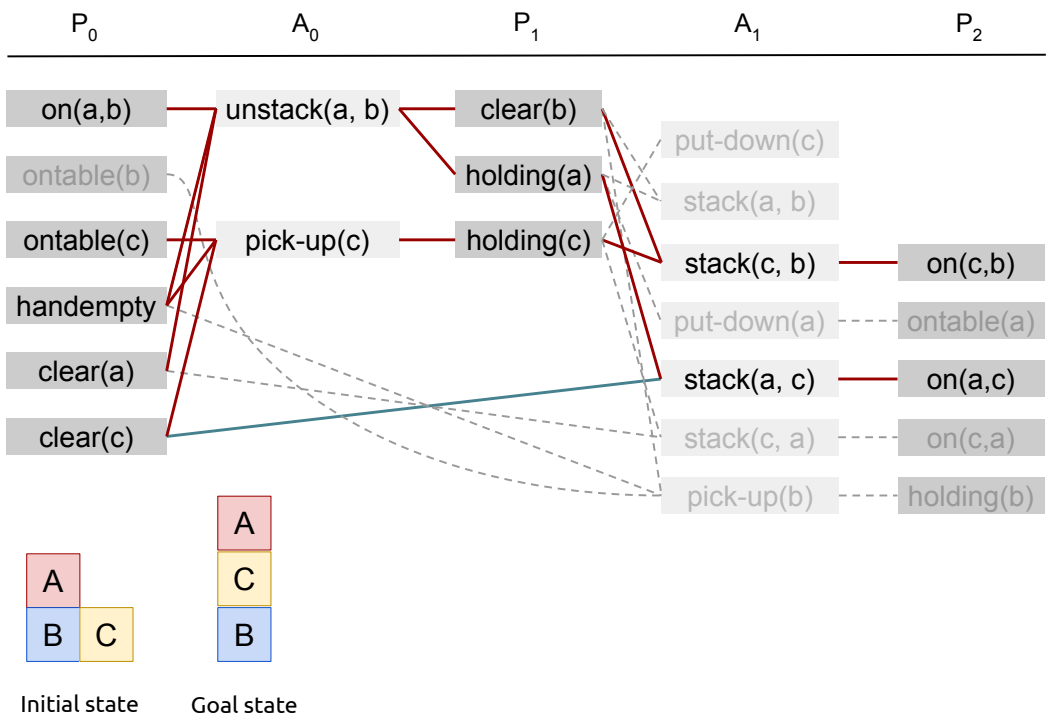

Figure 2.6: Plan extraction phase for the relaxed planning graph depicted in Fig. 2.5, assuming the goal is given by the set of atoms $\{on(a,c), on(c,b)\}$. Atom layer P_2 satisfies the goal, and hence the construction of the RPG need not proceed further. Action and atom supports chosen during the plan extraction phase that proceeds backwards from layer P_2 are highlighted, the rest are dashed and greyed out. The relaxed plan that results from the plan extraction is $\pi^+ = \langle \text{unstack}(a,b), \text{pick-up}(c), \text{stack}(c,b), \text{stack}(a,c) \rangle$

2.4.5 Informed Search Algorithms for Classical Planning

Within the planning as search paradigm, the use of heuristics that exploit information about the goal to guide the search has proven key to tackle the combinatorial explosion inherent to classical planning, particularly in *satisficing* planning, when any (possibly suboptimal) plan is sought. Here, we provide a brief overview of *best-first search*, one of the most common heuristic search schemas in classical planning (Korf, 1998; Russell and Norvig, 2009). Best-first search is not a concrete search algorithm, but a general algorithmic schema, the guiding principle of which is to expand nodes in the order given by a certain *evaluation function* f that estimates the cost of the cheapest path running through a certain search node. Nodes with lower value $f(n)$ are thus expanded with higher priority. The general schema is depicted in Fig. 2.7. All best-first algorithms iteratively select the node n from the *open list* with lowest value $f(n)$. If n is not a goal, it is expanded by generating all of its child nodes and placing them in the open list, unless they had already been expanded, in which case they are discarded, as there is no point in processing them again. This last check requires the maintenance of a *closed list* (typically a hash table) in order to keep track of all expanded nodes. Depending on the evaluation function $f(n)$ and on the type of optimality guarantees expected from the algorithm, it is sometimes necessary to update nodes which were already on the open list, or even to *reopen*

Algorithm: best_first_search(problem Π)

```
root := make_node(state=Π.s_0, parent=None, action=None)
open := priority queue with node root as only element
closed := ∅
while open is not empty do
    n := extract node with lowest f(n) value from open
    if Π.is_goal(n.state) then return make_solution(n)
    Add n.state to closed
    foreach action a in Π.applicable(n.state) do
        n' := make_node(state=Π.successor(n.state), parent=n, action=a)
        if n' is not in open nor in closed then insert n' into open
        else if n' is in open with higher g value then
            replace the node in open with n'
        else if n' is in closed with higher g value then
            reopen the node
return Failure
```

Figure 2.7: The best-first search algorithmic schema.

nodes on the closed list by placing them back in the open list. In the finite search spaces usually found in planning, all best-first search instantiations are *complete*, meaning that they will find a solution whenever one exists.

Different instantiations of the evaluation function $f(n)$ result in different concrete search algorithms; $f(n)$ usually combines the accumulated cost of reaching the node, denoted by $g(n)$, with the estimation of the cost to reach the goal given by the heuristic function, $h(n)$. In the classical A* algorithm (Hart et al., 1968), for instance, the evaluation function is set to $f(n) = g(n) + h(n)$. The plans returned by A* are provably optimal as long as the heuristic function h is *admissible*, i.e. $h(s)$ never overestimates the minimum cost to reach a goal state from s. An interesting variant of A* is Weighted A*, in which the evaluation function is set to $f(n) = g(n) + w \cdot h(n)$, where $w > 1$ is some positive *weight* that trades off solution quality for computational speed (Pohl, 1970).

More relevant to our work is *greedy best-first search* (GBFS, sometimes called *pure heuristic search*; Korf, 1998; Russell and Norvig, 2009), where the accumulated cost is ignored by setting $f(n) = h(n)$, and thus the algorithm always expands the node with lowest heuristic value. Greedy best-first search offers no optimality guarantees, but typically runs significantly faster than A* or other best-first variations. For this reason, GBFS is heavily used in the context of satisficing planning, although its behavior is less well understood than that of A* — a drawback often observed in practice being that GBFS can get stuck in large *plateaus* or node regions with equal heuristic value (Hoffmann, 2001, 2002, 2005). Several recent publications tackle this issue (Imai and Kishimoto, 2011; Xie et al., 2014b,c; Wilt and Ruml, 2015; Heusner et al., 2017).

Many recent planners combine different heuristics through the use of *multiple queues* (Helmert, 2006a), or use other techniques to enhance search performance such as

helpful actions (Hoffmann and Nebel, 2001a; Richter and Helmert, 2009), delayed node evaluation (Helmert, 2006a; Richter and Helmert, 2009), or the combination of different search algorithms, as is the case with the FF planner (Hoffmann and Nebel, 2001a), which chains a fast but incomplete Enforced Hill Climbing search with a greedy best-first search that is run only if the former is not able to find a plan.

2.4.6 Width-Based Search Algorithms

A recent approach to tackle the combinatorial explosion of planning as state-space search which is orthogonal to the use of heuristic evaluators is that of *width-based search* (Lipovetzky and Geffner, 2012). The basic idea in this type of search is to focus the search on those states that are in some sense novel. Lipovetzky and Geffner's original formulation is focused on propositional planning problems, but we here give a more general formulation. The precise definition of this general notion of *novelty* requires a couple of preliminary definitions.

Definition 2.13 (State factorization)**.** *Let s be a state of a search problem. A* state factorization *ϕ of size n is a function that maps s into a tuple $\phi(s) = \langle x_1, \ldots, x_n \rangle$, where $x_i \in D_i$, the i-th* feature of the factorization*, is often denoted by $\phi_i(s)$, and D_i is the domain of that feature.*

Definition 2.14 (Predecessor states)**.** *Let $s_0, s_1, \ldots$ be the ordered sequence of states generated by some search algorithm and, for any state $s \in S$, let $\gamma(s)$ be the* generation order *of s, i.e. the smallest i such that $s = s_i$, or ∞, if s is never generated by the search. The* set of predecessor states *of s in the search, denoted by $S_{\downarrow s}$, is $S_{\downarrow s} = \{s' \in S \mid \gamma(s') < \gamma(s)\}$.*

Based on the two previous definitions, we can now define the novelty of a state:

Definition 2.15 (Novelty of a state with respect to a search)**.** *Let ϕ be a state factorization of size n, and let $I \subseteq \{1, \ldots, n\}$ be a subset of the features of the factorization (identified by their indices). We say that a state s is I-novel in a search if for all previously-encountered states $s' \in S_{\downarrow s}$, there is always some feature $i \in I$ whose value in s' does not match its value in s, i.e. $\phi_i(s') \neq \phi_i(s)$. The novelty $w(s)$ of s is the size of the smallest set I such that s is I-novel. If no such subset exists (i.e. because s is a duplicate state already in $S_{\downarrow s}$), then $w(s) = n + 1$.*

Thus, a state s has novelty $w(s) = 1$ if there is one index $i \in \{1, \ldots, n\}$ such that s is the first state in the search in which the i-th feature has value $\phi_i(s)$. Otherwise, $w(s) = 2$ if there are two different indices $i, j \in \{1, \ldots, n\}$ such that s is the first state in the search in which both the i-th feature has value $\phi_i(s)$ and the j-th feature has value $\phi_j(s)$ *at the same time*, etc.

Many of the works that explore the application of width-based search to classical planning use the straight factorization that maps states to the values given by their state variables (Lipovetzky and Geffner, 2012, 2014, 2017a,b; Katz et al., 2017).[9] We will denote this *state-variable factorization* by ϕ^0. Other works such as (Lipovetzky

[9] Often, in propositional planning, where state variables are binary, only those features that have a *true* value are taken into account when defining the novelty of a state (Francès et al., 2017).

Algorithm: IW(problem Π, $k \in \mathbb{N}_1$)

```
if Π.is_goal(Π.s_0) then return ⟨⟩

root := make_node(state=Π.s_0, parent=None, action=None)
open := queue with node root as only element
while open is not empty do
    n := extract shallowest node from open
    foreach action a in Π.applicable(n.state) do
        n' := make_node(state=Π.successor(n.state), parent=n, action=a)
        if n' is not in open then
            if Π.is_goal(n'.state) then return make_solution(n')
(1)         if w(n'.state) <= k then insert n' into open
return Failure
```

Figure 2.8: The IW(k) Iterated Witdh search algorithm: A breadth-first search augmented with pruning of states with width higher than a fixed parameter k.

et al., 2015) or (Geffner and Geffner, 2015) use application-dependent factorizations. Although in Chapter 5 we briefly look at how these non-standard (and possibly domain-dependent) factorizations open up the prospect of improving the informativeness of the search, we restrict the following discussion of width-based algorithms to this ϕ^0 state-variable factorization, in which $\phi_i^0(s)$ is the value taken by the (possibly multivalued) i-th state variable of the state.

From a practical standpoint, checking if a certain state has novelty $w(s) \leq k$ can be computed in time and space $O(n^k)$, by storing for each value $i \in \{1, \ldots, k\}$ a global *novelty-i table* that contains all tuples of size i of the feature values seen in the search so far.

The IW(k) Algorithm

One straight-forward way to use the notion of novelty in a search is the IW(k) algorithm (Lipovetzky and Geffner, 2012), a simple breadth-first procedure that prunes those states that have novelty greater than k. Figure 2.8 shows a basic outline of the algorithm.[10] The closed list that is normally used in breadth-first search to detect duplicates is no longer necessary, since by definition duplicate states have novelty $n + 1 > k$ and will become pruned. Since the value of k is fixed, the computation of the novelty of the newly-generated state in line (1), actually boils down to computing whether the novelty of the state is either $w(s) \leq k$ or $w(s) > k$, which as we just saw has polynomial cost. On the other hand, the maximum number of states with novelty $w(s) <= k$ is upper-bounded by $O((n \cdot m)^k)$, where n is the number of state variables and $m = |D|$ is the cardinality of the largest state variable domain. For the binary domains that result from propositional planning, this can be seen to be $O(n^k)$. For fixed k, IW(k) thus runs in time and space polynomial, unlike breadth-first search, which requires time and space exponential in n. The pruning of IW(k) makes it in

[10] Given that the notion of novelty bears no relation with the notion of goal, IW(k) can be considered to all effects a blind search algorithm.

general an incomplete algorithm, unless $k = n$, in which case IW(n) is actually the breadth-first search algorithm. IW(k) can thus be seen as a way of approximating breadth-first search by trading off completeness for memory and speed.

Interestingly, Lipovetzky and Geffner (2012) show that IW(k) with $k \in 1, 2$ is able to find plans over many of the standard planning domains in low-polynomial time, *provided that the goal features a single atom only.* For many of those domains, one can prove theoretically that problems in the domains have a bounded and small *width* w, independent of the instance size. This implies that they can be solved optimally by running the IW(k) algorithm with $k = w$. A complete, iterative-deepening-like version of IW(k), called IW, works by calling IW(k) iteratively for increasing values of $k = 1, 2, \ldots$, which guarantees that a plan will eventually found if it exists. IW, however, is not optimal, as a call to IW(k) might succeed for a value of k lower than the actual width of the problem. In problems with multiple atomic goals, however, IW does not seem to be a particularly effective strategy to deal with the problem of finding the right serialization in which goals need to be achieved. The performance of extensions such as Serialized Iterated Width (SIW), where IW is called to achieve one atomic goal at a time, is significantly below that of state-of-the-art planners (Lipovetzky and Geffner, 2012). A more effective strategy for those cases seems to be best-first width search, which we introduce next.

Best-First Width Search (BFWS)

The IW(k) algorithm is simply an effective exploration strategy, but as we saw it makes no attempt at guiding the search towards goal nodes. This opens up the possibility of combining its effective exploration mechanism with the goal-directed guidance typically provided by standard planning heuristics. This has been recently attempted in (Lipovetzky and Geffner, 2017a,b; Katz et al., 2017), yielding state-of-the art results. An effective approach of this kind is *best-first width search* (BFWS; Lipovetzky and Geffner, 2017a), a greedy best-first search (see Fig. 2.7) where the node evaluation function f is a *lexicographic combination* of different metrics, the primary of which is novelty-based. This means that the evaluation function has form $f(n) = \langle w, f_1, \ldots, f_m \rangle$, where w is some measure of novelty, and $f_1, \ldots, f_m$ are arbitrary functions of the state in the search node, and the greedy best-first search underlying BFWS expands always the node in the open list with smallest value of w, breaking ties with f_1, then with f_2, and so on.

The best results for BFWS are obtained when the novelty is computed in a fine-grained manner over certain *partitions of the state space* S. The approach usually taken is to consider partitions induced by different heuristic functions. In general, for any set $F = \{f_1, \ldots, f_t\}$ of arbitrary functions over states, we can define a partition of S such that two states belong to the same equivalence class if and only if $f(s) = f(s')$ for all functions f in F.

Definition 2.16 (Novelty of a state given set of functions)**.** *Let ϕ be a state factorization of size n, and $F = \{f_1, \ldots, f_t\}$ a set of arbitrary functions defined over the set of states S. The* novelty $w_F(s)$ *of a state s* given the functions in F *is the size of the smallest subset of feature indices $I \subseteq \{1, \ldots, n\}$ such that for all predecessor states $s' \in S_{\downarrow s}$* that belong to the same equivalence class than s *(given $f_1, \ldots, f_t$), there is at least one feature index $i \in I$ such that $\phi_i(s') \neq \phi_i(s)$. If no such subset exists (i.e. because s is a duplicate state already in $S_{\downarrow s}$), then $w(s) = n + 1$.*

This essentially restricts Definition 2.15 so that the novelty of s is computed not with respect to *all predecessor states*, but *only with respect to those predecessor states in the same equivalence class than* s, i.e. with the same f_i-values than s.[11] The set F of functions used to partition the state space for the novelty computations is completely independent of the functions used with tie-breaking purposes to prioritize search nodes, although often the same functions can be used for both purposes. In BFWS(f_4), for instance (Lipovetzky and Geffner, 2017a), a greedy-best first search is used with evaluation function $f(n) = \langle w_F, h_{\mathrm{L}}, h_{\mathrm{FF}} \rangle$, where $F = \{h_{\mathrm{L}}, h_{\mathrm{FF}}\}$ contains the landmark heuristic h_{L} (Richter et al., 2008), and the `FF` heuristic h_{FF} (Hoffmann and Nebel, 2001a). This means that BFWS(f_4) partitions the state space into equivalence classes containing nodes with equal landmark and `FF` heuristic values, and the novelty of each generated state s is computed taking only into account those predecessor states that are in the same equivalence class as s. States with lower novelty w_F are expanded first, and ties are broken by prioritizing states with lower h_{L}, first, and states with lower h_{FF}, second.[12] Further variations of these ideas are discussed in Chapter 5.

2.5 State of the Art in Classical Planning

The state of the art in classical planning is usually evaluated and discussed through the International Planning Competitions, held in connection with the ICAPS conference at (somewhat uneven) time intervals since 1998, and articulated around different tracks (deterministic, temporal, probabilistic, learning, etc.), of which the most relevant to the work here presented is the deterministic track (McDermott, 2000; Long et al., 2000; Bacchus, 2001; Long and Fox, 2003; Hoffmann and Edelkamp, 2005; Gerevini et al., 2009; Helmert et al., 2008; Coles et al., 2012; Vallati et al., 2015a). Each competition usually evaluates planners on a set of benchmarks including domains both new and from previous competitions. Planners are allowed certain maximum time and memory limits, and a variety of evaluation criteria is used, including runtime, plan quality and coverage (i.e. number of problems solved under the given time and memory limits).

The first competitions (1998, 2000, 2002) saw heuristic search planners quickly become the dominating paradigm for satisficing planning, chiefly embodied in the `HSP` and `FF` planners (Bonet et al., 1997; Bonet and Geffner, 1998, 1999; Hoffmann and Nebel, 2001a). `HSP` implemented a form of hill-climbing driven by the h_{add} heuristic and coupled with restarts to escape from local minima. `FF`, in turn, implements a two-phase search, using the h_{FF} heuristic. The first phase is an incomplete *enhanced hill-climbing* search that focuses on *helpful actions* only, i.e. actions which are part of the relaxed plan that is obtained as a byproduct of the computation of h_{FF}. The second phase is a greedy best-first search driven by h_{FF} as well. Hoffmann and Nebel (2001b) conduct an empirical comparison between `HSP` and `FF`, and conclude that the superior performance of `FF` is largely due to the combined use of enhanced hill-climbing and helpful actions, rather than to the difference in heuristics. A different planner not based on the ideas of planning as heuristic search but that also made

[11] This partitioning is reminiscent to the type-based system discussed in (Lelis et al., 2013; Xie et al., 2014b), although their use is different.

[12] In practice, and due to the high cost of tracking and computing state novelty values higher than $k = 2$, the novelty values that are taken into account for the lexicographic ordering of nodes are usually discretized into $w(s) = 1$, $w(s) = 2$ or $w(s) > 2$; sometimes even only $w(s) = 1$, $w(s) > 1$.

use of Blum and Furst's planning graph and which showed top performance was LPG (Gerevini and Serina, 2002). LPG performs a form of randomized local search over a search space of partial plans defined by subgraphs of the planning graph, guided by heuristics that take into account the number of *inconsistencies* in any such partial plan.

Subsequent editions of the international planning competition saw the continued domination of heuristic search planning in the satisficing planning tracks. The top contender of the 2004 edition was the Fast-Downward planner (FD, Helmert, 2006a), which used an automatic transformation of propositional planning problems into a SAS^+-like multivalued representation, and combined the use of causal graph heuristic (Helmert, 2004) with different advanced techniques such as multiple search queues (Röger and Helmert, 2010), helpful actions (*preferred operators*) (Hoffmann and Nebel, 2001a; Richter and Helmert, 2009) and delayed or *lazy* node evaluation (Richter and Helmert, 2009; Helmert, 2006a). Over the years, Fast-Downward has grown into a heavily engineered and extensible codebase, offering a large number of search and heuristic strategies, and is often taken as the starting point from which novel ideas are tested. The prime example of this is the LAMA planner, winner of the 2008 competition, which builds on FD and crucially combines the h_{FF} heuristic with the novel landmark heuristic h_L (Richter et al., 2008), by using a greedy best-first search with multiple search queues; LAMA also uses the abovementioned helpful actions and delayed node evaluation techniques, and once a first solution is found, triggers a series of weighted A* search with decreasing weights, pruned by the cost of the best solution so far, to find plans with lower cost.

The last two competitions, in 2011 and 2014, have witnessed the raise of *portfolio* planners, which use several strategies to combine the often complementary strengths of existing techniques for increased performance. The winner of the 2011 competition, for instance, was the Fast Downward Stone Soup planner (Helmert et al., 2011), which uses all the available time to run independently, in sequence, a number of different planning strategies, among which the landmark-based BJOLP planner (Domshlak et al., 2011), the LM-cut heuristic (Helmert and Domshlak, 2009) and merge-and-shrink abstractions (Helmert et al., 2007). The exact strategies used and amount of runtime allowed to each strategy was statically determined from their respective performances on a set of training instances, which in this case included all suitable instances from previous competitions. The winners of the 2014 competition, the IBaCoP2 and IBaCoP2 portfolio planners (Cenamor et al., 2014), similarly select a subset of options among a pool of existing planners, but employ more sophisticated models to predict planner performance, based on performance on past competition instances as well. Portfolio techniques are used in several areas and make up an interesting field of research in themselves, but they do not provide particularly novel insights into the ways in which the classical planning problem can be addressed, and as such we will not be concerned with them in this work.

Other planners employing interesting approaches that have been developed over the years include YAHSP and its various extensions (Vidal, 2004a,b, 2011), PROBE (Lipovetzky and Geffner, 2011), Mercury (Katz and Hoffmann, 2014), Jasper (Xie et al., 2014a) and Madagascar (Rintanen, 2014). YAHSP uses polynomial-time delete-free relaxed plans as a lookahead mechanism, by considering how far one can go applying the relaxed plan as if it was a normal plan, and inserting that state in the search queue of a standard heuristic search algorithm. PROBE uses a more sophis-

ticated but still polynomial-time strategy to compute, without search, sequences of actions which in practice turn out to be correct plans quite frequently, and when they are not, can be used as a lookahead strategy in a similar fashion as in `YAHSP`. `Mercury` uses the ideas of partial delete relaxation embodied in the *red-black* planning approach (Katz et al., 2013), which roughly consists on ignoring only *some* of the delete effects, namely, those that affect certain variables of the problem, in order to compute the heuristic. `Jasper` aims at dealing more effectively with heuristic plateaus, i.e. large regions of the state space where the heuristic function is unable to discriminate between search nodes. To achieve that, it improves the `LAMA` planner with two mechanisms: a local search that is triggered whenever the global greedy best-first search is unable to produce improvements for a certain number of steps, and a *type system* that partitions search nodes according to different heuristic values and helps diversify the search (Xie et al., 2014b). Finally, `Madagascar` is a performant SAT-based planner that implements several enhancements to the classical reduction of planning to SAT, including more compact encodings that result in shorter horizons (Rintanen et al., 2006) and planning-specific SAT-solving heuristics (Rintanen, 2012).

As mentioned in Section 2.4.6, another recent development which is directly related to the work we present in this thesis and which achieves state-of-the-art performance is the best-first width search algorithmic schema (BFWS; Lipovetzky and Geffner, 2017a), which builds on the notion of novelty to perform an effective exploration of the state space. The BFWS(f_4) algorithm by Lipovetzky and Geffner presented above, for instance, solves 149 instances out of the 280 International Planning Competitions (IPC) instances on which it is tested, compared to 171 solved by `LAMA` or 198 by `Jasper` (Lipovetzky and Geffner, 2017a). Other BFWS variations work even better. BFWS(f_5), which will be discussed in Chapter 5, solves 192 instances, outperforming both `LAMA` and `Jasper`, while the *Dual-BFWS* strategy solves 225 instances, significantly above the 198 instances solved by the `IBaCoP2` portfolio planner. Interestingly, the polynomial flavor of novelty-k measures for low values of k can be additionally exploited to devise (incomplete) variations of the basic greedy-best first search in which nodes with novelty higher to a certain threshold are pruned. This turns out to be a surprisingly powerful search schema for standard classical planning benchmarks: Lipovetzky and Geffner (2017b) report that over a large benchmark base comprising 1676 instances from all International Planning Competitions, variations of this search schema solve as much as 1518 instances, compared to 1504 instances solved by `Mercury` or 1556 by `Jasper`. This is an astounding result, taking into account that these variations *run in polynomial time*.

2.6 Constraint Satisfaction and Satisfiability

Constraint satisfaction and Boolean satisfiability are both powerful frameworks to represent and solve a wide range of combinatorial problems. Constraint satisfaction is primarily concerned with the representation and solution of problems where an assignment of values to variables consistent with certain constraints is sought (Montanari, 1974; Mackworth, 1977), whereas satisfiability is concerned with finding models of propositional formulas (Biere et al., 2009). In this section we briefly define and discuss the satisfiability (SAT) and constraint satisfaction problems (CSP), with an emphasis on those concepts related to CSP that are related to the work we present in Chapters 3 and 4. For a more detailed account, we refer the interested reader to

(Dechter, 2003; Rossi et al., 2006; Biere et al., 2009; Russell and Norvig, 2009).

Definition 2.17 (Constraint Satisfaction Problem)**.** *A constraint satisfaction problem (CSP)* $\Pi = \langle X, D, C \rangle$ *consists of a finite set of* variables $X = \{x_1, \ldots, x_n\}$*, a* domain $D(x)$ *for each variable* $x \in X$*, containing the values allowed for that variable, and a set of* constraints $C = \{C_1, \ldots, C_m\}$*. Each constraint* $C_i = \langle S_i, R_i \rangle$ *is made up of a* scope S_i *and a* relation R_i*. The scope is a tuple of variables* $S_i = \langle x_1, \ldots, x_k \rangle$ *affected by the constraint, while the relation is a subset* $R_i \subseteq D_1 \times \cdots \times D_k$*. The* arity *of the constraint is the size of its scope.*

A complete assignment *is a mapping* σ *from each variable* x_i *to a value* $\sigma(x_i) \in D(x_i)$*. An assignment* σ satisfies *a constraint with scope* $\langle x_1, \ldots, x_k \rangle$ *and relation* R *if* $\langle \sigma(x_1), \sigma(x_2), \ldots, \sigma(x_k) \rangle \in R$*. A solution to a constraint satisfaction problem is a complete assignment* σ *that satisfies all the constraints of the problem.*

Example 2.18 (Vertex Coloring)**.** *As an illustration of the above definition, consider the* k-VERTEX-COLORING *problem, where we are given an undirected graph* $G = \langle V, E \rangle$ *and* k *colors* $c_1, \ldots, c_k$*, and are required to find a coloring of each one of the graph nodes so that no two adjacent nodes are painted with the same color. Assuming that* $V = \{v_1, \ldots, v_n\}$*, the problem can be cast as a constraint satisfaction problem* $\Pi = \langle X, D, C \rangle$ *with set of variables* $X = \{x_1, \ldots, x_n\}$*, each of which having domain* $D(x_i) = \{c_1, \ldots, c_k\}$*, and a set of constraints* C *that contains one constraint* $x_i \neq x_j$ *for each every* $(v_i, v_j) \in E$*. Each variable* x_i *is intuitively meant to represent the color assigned to vertex* x_i *in a particular vertex coloring; "*$x_i \neq x_j$*" can be read as an intensional representation of an actual constraint with scope* $\langle x_i, x_j \rangle$ *and relation* $\{\langle a, b \rangle \mid a \in D(x_i) \wedge b \in D(x_j) \wedge a \neq b\}$*. A possible solution to the problem* Π*, if it exists, will simply be an assignment of colors to each variable.*

Closely related to constraint satisfaction, the problem of Boolean *satisfiability* can be succinctly defined as a CSP with Boolean variables and clausal constraints. This definition, however, obscures the fact that SAT and CSP have vastly different origins and historical developments within the field of artificial intelligence, although recently the connections between both of them have begun to be explored (Walsh, 2000; Bacchus, 2007; Ohrimenko et al., 2009; Stuckey, 2010). The satisfiability problem consists in finding an assignment to a set of Boolean variables that satisfies a given propositional formula, typically given in conjunctive normal form:

Definition 2.19 (Boolean Satisfiability Problem)**.** *Let* ϕ *be a propositional formula over a set of variables* $X = \{x_1, \ldots, x_n\}$*. The* Boolean satisfiability problem *consists in finding whether there is a* truth assignment T*, i.e., a substitution* $T : X \mapsto \{\text{true}, \text{false}\}$*, such that the result of replacing the variables in* ϕ *as dictated by* T *evaluates to true according to the standard semantics of propositional logic. If* ϕ *is restricted to be in conjunctive normal form (CNF), i.e. is a conjunction of* clauses*, where a clause is a disjunction of literals, then the problem is often named* CNF-SAT*; if all clauses in the formula are restricted to have at most* k *literals, then the problem is named* k-SAT*.*

Both SAT and CSP are NP-complete problems.[13] Furthermore, there is a close relation between satisfiability and (bounded) planning: the problem of finding a plan

[13] SAT was indeed the first problem to be proven NP-complete (Cook, 1971; Karp, 1972; Levin,

of bounded polynomial size, also NP-complete, can be encoded as a SAT problem (Kautz and Selman, 1992, 1996) in a rather natural manner; as a matter of fact, this is precisely the approach followed by some state-of-the-art planners (Rintanen, 2012, 2014). Similar compilations of planning to constraint satisfaction have also been explored, although with less success (Do and Kambhampati, 2001; Nareyek et al., 2005). Both SAT and CSP have witnessed very significant progress over the last decades. Thanks to developments such as the DPLL algorithm or conflict-driven clause learning (Davis and Putnam, 1960; Marques-Silva et al., 2009), SAT solvers now are able to tackle problems with over a million of variables and several million clauses (Gomes et al., 2008).

Progress in CSP, on the other hand, has focused in the exploration of efficient constraint propagation methods for several types of constraints, of different notions of local consistency and of variable and value ordering heuristics, among others (Rossi et al., 2008). Most relevant to the subject of this thesis, a key concern in constraint satisfaction that is less pressing in Boolean satisfiability (since CNF, a normal form itself, is the widely-accepted, low-level representation of choice for SAT instances), is *modeling* (Freuder, 1999; Smith, 2006). Issues related to the computational value of different representations of the same problem have been thoroughly studied, and powerful constraint modeling languages (e.g. Savile Row, Essence, Zinc, MiniZinc) are actively being developed and improved (Frisch et al., 2007; Nethercote et al., 2007). In the remainder of this section, we focus on some other constraint satisfaction concepts related to the work presented in this thesis.

A key idea for scaling up efficiently in the face of the worst-case theoretical hardness of the constraint satisfaction problem is that of *local consistency*, which allows standard search methods to be combined with a form of inference called *constraint propagation*. For that, the problem constraints are used not only passively (i.e., to check whether a given assignment satisfies them), but also actively, to *prune* the set of possible values that is considered feasible for each variable at any given moment during the search. For instance, assume we have two variables x and y both with domain $\{1, 2, 3\}$, and a constraint $x = y + 1$. We can infer from that constraint that value 1 will never be possible for variable x, and likewise for value 3 and variable y. Those values are said to be locally inconsistent with the constraint, and can be safely pruned from the domains under consideration without affecting the set of solutions of the problem. This local consistency checks can normally be performed efficiently, and their cost is usually largely compensated by the reduction in the amount of search necessary to find a solution.

Different notions of local consistency are possible, each with different costs and pruning power. A variable x_i, for instance, is said to be *node consistent* with respect to a unary constraint iff all values in $D(x_i)$ satisfy the constraint. In turn, x_i is said to be *arc consistent* with respect to a binary constraint with scope $\langle x_i, x_j \rangle$ iff for every value $v \in D(x_i)$, there is one value $v' \in D(x_j)$ such that $\langle v, v' \rangle$ satisfies the constraint. Arc consistency can be generalized to higher-arity constraints (*generalized arc consistency*). A CSP is arc consistent if each of its variables is arc consistent with respect to all constraints that involve it. Enforcing arc consistency on a CSP usually involves iterating over all variables and pruning inconsistent values from their domains until either a fixpoint is reached and the problem is arc consistent, or some domain becomes empty, in which case the problem is unsolvable (Mackworth, 1977).

1973).

In general, constraints can be represented *extensionally*, by listing all tuples contained in its relation R, or *intensionally*, by means of some procedure that encodes the characteristic function of R, i.e. is able to tell whether a given tuple belongs to R or not. Unary and binary constraints are the most common in constraint modeling, but another key aspect of the field is the use of global constraints. A *global constraint* is a constraint with arbitrary arity that usually affords an elegant and compact way of modeling some recurrent concept. A quadratic number of binary constraints $x_i \neq x_j$, for instance, can be conveniently encoded with a single global $\texttt{alldiff}(x_1, \ldots, x_n)$ constraint, that enforces that no two variables take the same value. Other popular global constraints include different sorts of cardinality, packing and scheduling constraints (Régin, 2011).

Global constraints are advantageous not only because they allow more compact and elegant models, but because at the same time they allow a more effective use of the structure of the problem. This is done through the use of specialized propagation algorithms that take into account the structure and semantics of each particular constraint. Enforcing arc consistency on an arbitrary constraint of arity k has worst-case complexity $O(kd^k)$, where d is the maximum size of the involved domains (Rossi et al., 2008). In contrast, for a simple constraint such as $x \leq y$, it can be easily established that all values that lie outside the range $[\min(D(x)), \max(D(y))]$ are arc inconsistent, and thus can be pruned more much more efficiently. For a more complex constraint such as $\texttt{alldiff}(x_1, \ldots, x_n)$, not only arc consistency can be established in $O(k^2 d)$ (van Hoeve, 2001), but equally important, enforcing arc consistency on such an all-different constraint achieves a stronger pruning than enforcing it over its binary decomposition into a quadratic number of inequality constraints $x_i \neq x_j$. Indeed, arc consistency over the binary constraints will find no inconsistency when all variables have domain $D(x_i) = \{0, 1\}$, yet it is known that such constraints cannot be jointly satisfied if $|\cup_{i=1}^{n} D(x_i)| < n$.

Chapter 3

Planning with Function Symbols

3.1 Motivation

The main contribution of Functional STRIPS (FSTRIPS) is the extension of the STRIPS language with function symbols, which might appear a minor change, but entails significant consequences for both modeling and problem solving (Geffner, 2000). As shown by Geffner, function symbols allow encodings with fewer ground actions and result in state representations that are closer to those of specialized solvers. FSTRIPS representations are often more compact and *readable* than their STRIPS counterparts. Figure 3.1, for instance, presents a FSTRIPS encoding of the classical blocks-world, where a single action schema $move(b, x)$ suffices to capture the dynamics of the problem. Standard STRIPS encodings, in contrast, require different action schemas to represent moves of blocks from and to the table, or require extra fluents to encode moves as a sequence of *pick* and *place* operations.

Easing the modeling task is but one of the objectives of modeling languages. Another is capturing *problem structure* in a way that can be exploited by solvers (Rintanen, 2015). In this chapter, we argue that function symbols are convenient for both objectives. The use of functions captures some key constraints that are usually obscured by propositional encodings, but which are useful to derive more informed heuristics. Even if a planner designer is not willing to offer support for functions, the additional expressiveness allowed by functions remains attractive, and problems making use of them can be automatically translated into function-free encodings. This, however, does not mean that dealing with the translation is bound to be computationally as easy as dealing with the original encoding (Bäckström, 1994), which is precisely one of the main messages of this chapter.

An example. To illustrate, consider a simple planning problem involving a set of integer variables $x_1, \ldots, x_n$ plus two actions per each variable x_i: one increases the value of x_i by one, the other one decreases it also by one, provided the value remains within some interval $[0, m]$. Initially, all variables have value 0, and the goal is to achieve the inequalities $x_1 < x_2 < x_3 < \ldots < x_n$. We name this problem COUNTERS. Because of the use of integers and arithmetic relations, it is not straightforward to represent COUNTERS in a propositional language. A possible PDDL encoding is shown in Fig. 3.2.[1] In this encoding, there is one atom $val(x_i, k)$ representing each equality

[1] See also the next chapter (Fig. 4.1) for a simpler encoding in (first-order) PDDL with existential quantification.

```
types place, block subtype of place

objects
    T: place
    b_1, ..., b_n: block

predicate clear(x: place)

function loc(b: block): place

action move(b: block, x: place)
  prec b ≠ x ∧ loc(b) ≠ x ∧ clear(b) ∧ clear(x)
  effs loc(b) := x
       clear(loc(b))
       x ≠ T → ¬clear(x)

init loc(b_1) = T, loc(b_2) = b_1, ... clear(b_2), ...
goal loc(b_1) = b_2 ∧ loc(b_2) = T ∧...
```

Figure 3.1: Functional STRIPS encoding of the classical blocks-world domain (simplified syntax). The encoding requires one single action schema *move* that moves in one step any given block b to the given place x, which might be another block or the table, represented here with the constant T.

$x_i = k$, and one atom $\texttt{lt}(x_i, x_j)$ representing each (strict) inequality $x_i < x_j$. Keeping the intended semantics of the latter requires a linear number of conditional effects. The goal is expressed by the conjunction of all atoms $\texttt{lt}(x_i, x_{i+1})$, for $i \in [0, n-1]$.

From a modeling point of view, COUNTERS illustrates how Functional STRIPS allows simpler encodings. Figure 3.3 shows a FSTRIPS encoding of the same problem. Each variable x_i is represented by the term $val(c_i)$ (a multivalued state variable). No other state variables are needed, and no logical symbols besides *val* and the constants $c_1, \ldots, c_n$, as the goal formula uses the standard symbol "<". The increase of a variable can be directly represented with a single effect `val(`c`) := val(`c`) + 1`, fully supported by the language. The encoding has $2n$ ground actions (including *increment* and *decrement* actions), whereas the propositional PDDL encoding has $2nm^2$ ground actions, of which $2nm$ are potentially applicable in some state.

From a computational point of view, COUNTERS illustrates the shortcomings of propositional heuristics such as those based on the Relaxed Planning Graph (RPG), introduced in Chapter 2. It is easy to see that the atom layer P_1 of the RPG for the propositional encoding of COUNTERS makes true each of the goal atoms $\texttt{lt}(x_i, x_{i+1})$, because only a single increment is needed to make each such atom true. The initial state s_0 of the problem thus has an $h_{\max}$ heuristic value of 1, and an h_{FF} value of $n-1$; the shortest plan for the problem, however, has $h^*(s_0) = 1+2+\cdots+n-1 = n(n-1)/2$ steps. What is interesting is not the failure of the heuristics to provide a better approximation (they are, after all, heuristics), but the fact that their inaccuracy is due to the the assumption implicit in the RPG that goal atoms such as $\texttt{lt}(x_1, x_2)$ and $\texttt{lt}(x_2, x_3)$ are independent, while they are not. Their dependence, however,

```
types counter, int

objects
    x_1, ..., x_n: counter
    i_1, ..., i_m: int

predicates
    val(x: counter, v: int)
    lt(x: counter, y: counter) ;; i.e. val(x) < val(y)
    S(v_0: int, v_1: int) ;; The successor relation, v_1 = v_0 + 1

action increment(x: counter, v_0: int, v_1: int)
  prec val(x, v_0) ∧ S(v_0, v_1)
  effs ¬val(x, v_0)
       val(x, v_1)
       ∀y ∈ counter (x ≠ y ∧ val(y, v_0)) → lt(y, x)
       ∀y ∈ counter (x ≠ y ∧ val(y, v_1)) → ¬lt(x, y)

init val(x_1, 0), val(x_2, 0), ..., val(x_n, 0)
goal lt(x_1, x_2) ∧ ... ∧ lt(x_{n-1}, x_n)
```

Figure 3.2: Fragment of a PDDL encoding of the COUNTERS domain (simplified syntax), where n variables take values in the range $[0..m]$; the *increment* action increases the value of a single variable by one.

```
type counter

objects c_1, ..., c_n: counter

function val(c: counter): int[0..m]

action increment(c: counter)
  prec val(c) < m
  effs val(c) := val(c) + 1

init val(c_1) = 0, val(c_2) = 0, ..., val(c_n) = 0
goal val(c_1) < val(c_2) ∧ ... ∧ val(c_{n-1}) < val(c_n)
```

Figure 3.3: Fragment of a FSTRIPS encoding of the COUNTERS domain (simplified syntax), for comparison with the PDDL encoding in Fig. 3.2.

is represented, in the propositional encoding, in the delete effects of the *increment* and *decrement* actions, and is therefore lost in the relaxation. If we analyze this type of delete-free relaxation heuristics over more expressive encodings making use of numeric (i.e. multivalued) state variables (Rintanen and Jungholt, 1999; Hoffmann, 2003; Coles et al., 2008; Helmert, 2009; Holte et al., 2014), a different picture emerges, in which the delete-free nature of the heuristic is not the only responsible for its lack of accuracy, but the loss of logical structure in the move from first-order atoms such as $val(c_1) < val(c_2)$ to propositional atoms such as $\texttt{lt}(x_1, x_2)$ also plays a major role.

3.2 Overview of Results

In this chapter, we analyze the relaxed planning graph from a logical point of view, to gain a better understanding of the sources of inaccuracy of RPG-based heuristics. These are related not only to the absence of delete effects, but also to the assumption that atoms in a conjunction are independent, which is justified only in a propositional setting. The key idea is how to determine whether a formula over the problem language *is reachable in a certain layer* of the RPG. Accounts of the RPG inspired by the propositional nature of its original formulation focus mostly on formulas that are conjunctions of atoms, and assume that they are reachable in a layer if each of the atoms is reachable in that layer. This is tractable and works well for propositional formulas, but is too weak an inference for first-order formulas: with this mechanism, a formula such as $x < 0 \wedge x > 0$ cannot be detected as unsatisfiable.

We present an alternative, first-order account of the RPG which circumvents this independence assumption and embodies a stronger inference mechanism. Our account explicitly identifies each layer of the RPG as *encoding* a set of *reachable first-order interpretations over the problem language*. A formula is then considered reachable in some RPG layer if at least one of the interpretations represented by that layer satisfies the formula *according to the standard notion of first-order satisfiability*. This is an elegant approach, and we show that it can be easily mapped into a constraint satisfaction problem. The downside is that the operation becomes worst-case intractable, but we empirically show that in practice the constraint satisfaction problems are simple enough to be efficiently solvable,[2] and that in many cases (which we illustrate) the stronger inference results in more informed heuristics, whose use in standard search algorithms pays off for their increased cost. Additionally, we show how the inference mechanism in our first-order relaxed planning graph can be approximated in polynomial time by using standard constraint satisfaction techniques, trading off informativeness for speed. The first-order RPG that we present and its mapping into a constraint satisfaction problem allow for a number of language extensions, such as the use of global constraints in the problem definition, which are interesting both from a modeling and from a computational point of view. We also illustrate this.

While the importance of more expressive planning languages for modeling is well-known (Rintanen, 2006, 2011; Gregory et al., 2012; Rintanen, 2015; Ivankovic and Haslum, 2015), our emphasis is mainly on the computational value of such extensions, and their use for understanding the limitations and possible elaborations of current heuristics. Some of the language extensions that we consider, however, such as the use of *global constraints* (van Hoeve and Katriel, 2006) are novel in the context of

[2] Rintanen (2006) makes a similar argument.

planning and interesting on their own. Our technique also offers support for the use of fixed symbols whose denotation is given *intensionally*, by means of external procedures implemented in some programming language. This feature was already present in the first definition of Functional STRIPS (Geffner, 2000) and has recently gained attention again under the name of *semantic attachments* (Dornhege et al., 2009; Bernardini et al., 2017). Yet, while the planning language accommodates global constraints and semantic attachments, and the planning heuristics accommodate forms of constraint propagation, we hope to show that these are not add-ons but *pieces that fall into place from the logical analysis of the language and the heuristic computation.*

The remainder of the chapter is organized as follows. In Section 3.3 we define a generalization of the RPG to first-order languages that follows the so-called value-accumulating semantics (Gregory et al., 2012; Katz et al., 2013) and is based on the assumptions of *monotonicity* and *decomposability*. In Section 3.4 we propose an alternative, first-order relaxed planning graph, where layers of the graph are understood in terms of sets of reachable first-order interpretations, and where the assumption of decomposability is dropped. We give the technical details of the CSP-based computation of this first-order RPG in Section 3.5, and in Section 3.6 discuss how constraint satisfaction techniques can be used to obtain an approximate, polynomial version. In Section 3.7 we present some interesting language extensions easily supported by the first-order RPG, and in Section 3.8 we evaluate experimentally all of the previous ideas. We conclude the chapter in Section 3.9 with a discussion of our contribution and of related work. Most of the work presented in this chapter has been previously published in (Francès and Geffner, 2015; Ferrer-Mestres et al., 2015; Francès and Geffner, 2016a).

3.3 Value-Accumulating Relaxed Planning Graph

Heuristics based on the relaxed planning graph, such as $h_{\max}$ and h_{FF}, can be generalized to languages featuring finite-domain, multivalued state variables plus arbitrary formulas in a rather straight-forward manner, through the so-called *value-accumulating semantics* (Hoffmann, 2003; Gregory et al., 2012; Katz et al., 2013; Ivankovic et al., 2014).[3] In the value-accumulating relaxation, each time an applicable action reaches a new value for a certain state variable, this value is *accumulated* into a set of previously-reached values. Thus, each propositional layer P_k of the relaxed planning graph encodes a set x^k of values that are *reachable in k steps* for each state variable x. This preserves the key property of monotonicity, because sets x^k grow monotonically as actions supporting new values for x become applicable.

The value-accumulating semantics is often discussed in the context of multi-valued languages where atoms are restricted to have the form $x = c$ or $x \neq c$, with x being a variable and c an integer value. In that context, an atom $x = c$ is said to be satisfied in RPG layer P_k if $c \in x^k$, i.e. if c is one of the values that have been reached in

[3] Hoffmann (2003) is chiefly concerned with the generalization of the delete-free relaxation to problems featuring numeric variables and a limited number of arithmetic expressions over them; Katz et al. (2013) and Ivankovic et al. (2014) also discuss numeric variables, but they limit their usage to atoms of the form $x = v$, where x is a variable and v an integer value. In contrast, Gregory et al. (2012) presents a generalization of delete-free relaxation to any sort of (finite-domain) data type, which is valid not only for integer and real variables but also for other data types such as sets, intervals, etc. for which some kind of *value-accumulation* operation can be defined.

that layer; its negation $x \neq c$ is satisfied in that same layer if there is some element $c' \neq c$ such that $c' \in x^k$. Such notion of satisfiability can then be used to define the *sets* $\phi^k \subseteq \{\top, \bot\}$ *of possible truth values of arbitrary formulas* ϕ and from them, the sets of possible values x^{k+1} for the next propositional layer P_{k+1}. We next define a generalization of this setting to arbitrary variable-free terms, atoms and formulas, in the context of Functional STRIPS,[4] which basically requires to define how such entities are considered as *reachable* in any given layer of the RPG.

Definition 3.1 (Reachable Denotations of a Term)**.** *Let P be a FSTRIPS problem with* variable-free *language $\mathcal{L}(P)$, and t a term over $\mathcal{L}(P)$. The set t^k of possible denotations of t in atom layer P_k of the RPG relaxed planning graph can be inductively defined as follows:*

- *Assume t is a constant c. If its denotation is* fixed*, then $c^k = \{c^*\}$; if it is* fluent*, then by definition, c is a state variable x, in which case c^k is directly given by the RPG layer, $c^k = x^k$.*
- *If t is a term $f(t_1, \ldots, t_m)$, $m > 0$, then*

$$[f(t_1, \ldots, t_m)]^k = \begin{cases} \bigcup_{c_1 \in t_1^k, \ldots, c_m \in t_m^k} f^*(c_1, \ldots, c_m), & \text{if } f \text{ is fixed} \\ \bigcup_{x \in \mathcal{V}(f, t_1^k, \ldots, t_m^k)} x^k, & \text{if } f \text{ is fluent} \end{cases}$$

where $\mathcal{V}(f, t_1^k, \ldots, t_m^k) \subseteq \mathcal{V}(P)$ is the set of all state variables that can be derived from the symbol f applied to values from each of the sets t_i^k:

$$\mathcal{V}(f, t_1^k, \ldots, t_m^k) = \bigcup_{c_1 \in t_1^k, \ldots, c_m \in t_m^k} \underline{f(c_1, \ldots, c_m)}^5$$

Definition 3.2 (Reachable Truth Values of a Formula)**.** *Let P be a FSTRIPS problem with* variable-free *language $\mathcal{L}(P)$, and ϕ a formula over $\mathcal{L}(P)$. The set ϕ^k of possible truth values of ϕ in atom layer P_k of the RPG can be inductively defined as follows:*

- *Assume ϕ is a nullary relation symbol p. If it is a fixed symbol, then $p^k = \{p^*\}$; if it is fluent, by definition it is a (relational) state variable x, in which case p^k is directly given by the RPG layer, $p^k = x^k$.*
- *Assume ϕ is an atom $R(t_1, \ldots, t_m)$. If R is a fixed symbol, then $[R(t_1, \ldots, t_m)]^k$ contains the value $\top$ ($\bot$, respectively) iff there are values $c_1 \in t_1^k, \ldots, c_m \in t_m^k$ such that $\langle c_1, \ldots, c_m \rangle \in p^*$ (respectively, $\langle c_1, \ldots, c_m \rangle \notin p^*$).*

  If R is fluent, then $[R(t_1, \ldots, t_m)]^k$ contains the value $\top$ ($\bot$, respectively) iff there are values $c_1 \in t_1^k, \ldots, c_m \in t_m^k$ such that $\top$ ($\bot$, resp.) is contained in the set x^k of reachable truth values for the (relational) state variable $x \equiv \underline{p(c_1, \ldots, c_m)}$.
- *If ϕ is a disjunction, conjunction or negation involving atoms p and q, then*

  1. *$\top \in [\neg p]^k$ if $\bot \in p^k$;*

[4] Support for existential variables is discussed in Chapter 4.

[5] By $\underline{f(c_1, \ldots, c_m)}$ we refer to the state variable made up by symbols f, c_1, ..., c_m, not to any actual value, see Section 2.3.6.

2. $\top \in [p \wedge q]^k$ *if* $\top \in p^k$ *and* $\top \in q^k$;
3. $\top \in [p \vee q]^k$ *if* $\top \in p^k$ *or* $\top \in q^k$.
4. $\bot \in [\neg p]^k$ *if* $\top \in p^k$;
5. $\bot \in [p \wedge q]^k$ *if* $\bot \in p^k$ *or* $\bot \in q^k$;
6. $\bot \in [p \vee q]^k$ *if* $\bot \in p^k$ *and* $\bot \in q^k$.

We can now define the generalization of the value-accumulating RPG to FSTRIPS:

Definition 3.3 (Value-Accumulating Relaxed Planning Graph for FSTRIPS)**.** *Let P be a FSTRIPS problem with state variables $x_1, \ldots, x_n$. The value-accumulating relaxed planning graph RPG(P, s) of P built from state s is a succession $P_0, A_0, P_1, A_1, \ldots$ of interleaved* atom *(P_i) and* action *(A_i) layers, such that:*

- *Atom layer $P_0 = \langle x_1^0, x_2^0, \ldots, x_1^0 \rangle$ contains, for each state variable x_i, the set $x_i^0 = \{x_i^s\}$ with the value of x_i in the state from which the RPG is computed.*
- *Action layer $A_i = \left\{o \in O \mid \top \in [pre(o)]^i\right\}$ contains all actions considered* applicable *in the previous atom layer P_i, i.e. such that the truth value $\top$ is reachable for the precondition of the action.*
- *Atom layer $P_{i+1} = \langle x_1^{i+1}, x_2^{i+1}, \ldots, x_1^{i+1} \rangle$ contains, for each state variable x_i, the union of x^i with the set of possible values for x_i that are supported by the (possibly conditional) effect of some action in $o \in A_i$, called the* supporter *of those values. A conditional effect $\phi \to f(t_1, \ldots, t_m) := w$ of a supports value v of x in P_k iff $\top \in \phi^k$, $v \in w^k$, and x is the state variable $\underline{f(c_1, \ldots, c_m)}$ for some values $c_1 \in t_1^k, \ldots, c_m \in t_m^k$.*

This finishes the definition of the sequence of propositional layers P_0, ..., P_k that make up the relaxed planning graph for a given problem P and state s. When computing the heuristics $h_{\max}$ and h_{FF}, the computation can be stopped in the first layer P_k where the goal formula G *is true*, i.e. where $\top \in G^k$, or whenever a fixed point has been reached without rendering the goal true, i.e. $x^k = x^{k+1}$ for all state variables x in the problem. In the second case, $h_{\max}(s) = h_{\text{FF}}(s) = \infty$ as one can show that there is no plan for P from s. In the first case, $h_{\max}(s) = k$, and a relaxed plan $\pi_{\text{FF}}(s)$ can be obtained backward from the goal by applying the standard RPG extraction algorithm: keeping track of the state variables x and values $v \in x^k$ that make the goal true, the actions o and effects $\phi \to f(t) := w$ supporting such values first, and iteratively, the variables and values that make $pre(o)$ and ϕ true. The heuristic $h_{\text{FF}}(s)$ is given by the length of the plan $\pi_{\text{FF}}(s)$.

Example 3.4. *Let P be the Functional STRIPS instance of the above-defined* COUNTERS *problem (Fig. 3.3) with 3 integer variables x_1, x_2, x_3, with domain ranges $[0, 3]$. Recall that the goal formula is $(x_1 < x_2) \wedge (x_2 < x_3)$, the initial state s_0 is given by $x_1 = 0, x_2 = 0, x_3 = 0$, and actions increment or decrement each variable within the $[0, 3]$ range. In the relaxed planning graph computed from the s_0, we have a first atom layer P_0 where for all i, the set of reachable values for each state variable is $x_i^0 = \{0\}$. The first action layer A_0 contains all actions o such that the truth value $\top$ is reachable for $pre(o)$ in layer P_0. This includes all* increment *actions, since $[x_i < 3]^0 = \{\top\}$: the state variable x_i has one single value 0 reachable in the first layer, and likewise for the fixed constant symbol 3; they together clearly belong to*

the denotation of the fixed symbol "<". Following a similar reasoning, decrement *actions are not applicable in layer* P_0.

Now, the sets of reachable values in the next atom layer are $x_1^1 = x_2^1 = x_3^1 = \{0, 1\}$. *As it turns out, this implies that* $\top \in [x_1 < x_2]^1$ *as there are values* 0 *and* 1 *in* x_1^1 *and* x_2^1 *such that* $0 < 1$. *Similarly,* $\top \in [x_2 < x_3]^1$, *so we get* $\top \in G^1$.

One can easily see that the defined relaxation generalizes to a first-order language such as FSTRIPS, yet the variable x_1 can have *both values* 0 *and* 1 *at the same time*, using 0 to make the first goal true and 1 to make the second goal true. Indeed, in this relaxation, *self-contradictory goals like* $x_1 = 0 \wedge x_1 = 1$ *are achievable in one step as well.* Thus, applying this *value-accumulating semantics* to our COUNTERS problem, we find that each of the atoms $x_i < x_{i+1}$ is true in layer P_1, thus yielding the same heuristic values as in the propositional encoding. In the next section, we offer an alternative relaxation based on the notion of first-order logical interpretation that addresses this shortcoming.

3.4 First-Order Relaxed Planning Graph

In the above discussion, it is possible to see that the inaccuracy of the heuristic is not a result of the delete-free relaxation itself, but rather of the way in which the value-accumulating semantics have been defined. While it is correct to regard *each* of the atoms $x_i < x_{i+1}$ as true in layer P_1, where all variables have domain $\{0, 1\}$ and hence there are values for x_i and x_{i+1} that satisfy $x_i < x_{i+1}$, it is not correct to regard the *conjunction of all of them* as true, since there are no values for the state variables in the domains $D(x_i) = \{0, 1\}$ that can satisfy all the atoms $x_i < x_{i+1}$ *at the same time* (assuming $n > 2$).

Monotonicity and Decomposability. From a logical perspective, the value-accumulating semantics is too weak because it makes *two simplifications*, not just one. The first is *monotonicity*, by which the variable domains $D(x)$ grow monotonically as new values for variable x become reachable, in line with the notion of "delete-relaxation". The second is *decomposability*, by which a *conjunction* of atoms is regarded as *true* in a propositional layer whenever *each one* of the atoms in the conjunction is true. Like monotonicity, decomposability is *not* true in general.

A way to understand this to see each propositional layer of the relaxed planning graph as the encoding of *a set* $\mathcal{I}_k$ *of possible first-order interpretations over the language.* For a conjunction of atoms $p \wedge q$, it is possible that one interpretation in $\mathcal{I}_k$ makes p true, a second interpretation makes q, and yet no interpretation makes the two atoms true at the same time. Decomposability is the assumption that if $\mathcal{I}_k$ contains interpretations that satisfy *each* of the atoms in a conjunction, it will also contain interpretations that satisfy *all* of the atoms in that conjunction. This is actually a valid assumption when no two atoms in the conjunction involve the same state variable, e.g. in conjunctions such as $x_1 > 3 \wedge x_2 < 2$, or $\mathit{clear}(b) \wedge \mathit{ontable}(b)$, where the conjuncts involve different state variables. This class of conjunctions indeed subsumes the standard STRIPS language fragment handled by most classical planners, where preconditions, conditions, and goals are conjunctions of propositional atoms such as $\mathit{on}(a, b)$ that can be seen as (binary) state variables. In such conjunctions, no state variable is mentioned more than once. This language fragment also subsumes

the *restricted numeric planning tasks* in `Metric-FF` where atoms can contain at most one numeric variable, and hence can be of the form $x = c$ or $x > c$, where c is a constant, but not of the form $x > y$, where both x and y are numeric state variables.[6]

However, in general, the fact that different values are all regarded as possible for a certain state variable in layer P_k of the relaxed planning graph does not imply that they are *jointly* possible. A possible way to retain monotonicity in the construction of the RPG while while removing the assumption that a state variable can take several values at the same time is to use an alternative, first-order semantics of the RPG, which we define next.

Definition 3.5 (First-order Interpretations Corresponding to a Relaxed Planning Graph Layer)**.** *Let P be a FSTRIPS problem with state variables $x_1, \ldots, x_n$, and P_k the propositional layer of a relaxed planning graph built from some problem state. As we saw in Section 2.3.6, a FSTRIPS state can be alternatively seen as a complete assignment of values to state variables or as a first-order interpretation over the language $\mathcal{L}(P)$ of the problem. The set $\mathcal{I}_k$ of first-order interpretations that are reachable in the layer P_k contains all those interpretations derived from the complete assignments given by the Cartesian product $x_1^k \times \cdots \times x_n^k$.*

With this, a different notion of (relaxed) reachability in any RPG layer can be defined in a concise and elegant manner.

Definition 3.6 (Formula Reachability in the First-Order RPG)**.** *Let P be a FSTRIPS problem with language $\mathcal{L}(P)$. A formula ϕ over $\mathcal{L}(P)$ is satisfiable in propositional layer P_k iff* there is an interpretation $\mathcal{M} \in \mathcal{I}_k$ *that satisfies ϕ.*

We call this alternative definition of the relaxed planning graph the *First-Order Relaxed Planning Graph* (FOL-RPG). This definition affects the contents as well as the computation of the RPG, keeping the monotonicity assumption, but dropping that of decomposability. In the FOL-RPG, a functional conditional effect $\phi \rightarrow f(\bar{t}) := w$ of an action o supports the value v of state variable x in P_k iff there is an interpretation $\mathcal{M} \in \mathcal{I}_k$ such that $\mathcal{M} \vDash (pre(o) \wedge \phi)$, $w^{\mathcal{M}} = v$, and $f(\bar{t})$ *resolves* under $\mathcal{M}$ to the state variable x, i.e. f is a fluent symbol and $x \equiv \underline{f(\bar{t}^{\mathcal{M}})}$. Analogously, a relational conditional effect $\phi \rightarrow R(\bar{t})$ ($\phi \rightarrow \neg R(\bar{t})$ respectively) supports value $\top$ ($\bot$) for the binary state variable x, iff there is an interpretation $\mathcal{M} \in \mathcal{I}_k$ such that $\mathcal{M} \vDash (pre(o) \wedge \phi)$ and $x \equiv \underline{R(\bar{t}^{\mathcal{M}})}$.

Heuristics. We denote by $h^{FO}_{\max}$ and h^{FO}_{FF} the heuristics that can be computed from the FOL-RPG. To compute them, the construction of the RPG stops at the first layer P_k where the goal formula ϕ_G is satisfiable, i.e. where there is some interpretation $\mathcal{M} \in \mathcal{I}_k$ such that $\mathcal{M} \vDash \phi_G$, or when a fixed point is reached without rendering the goal true. We distinguish these heuristics from the standard $h_{\max}$ and h_{FF}, as they behave in a different way, produce different results, and have different computational

[6] In `Metric-FF`, two atoms in a conjunction can actually involve the same state variable, provided that they do not involve any other state variable. That is, a conjunction can feature the atoms $x > 1$ and $x < 5$, yet such conjunctions, as we will see can be easily compiled into a single atom such as $between(x, 1, 5)$. Indeed, decomposability is true on conjunctions of such atoms as well provided that that such conjunctions are logically consistent.

cost. From a semantic standpoint, inconsistent goals like $(x < 0 \wedge x > 0)$ produce infinite $h^{FO}_{\max}$ and h^{FO}_{FF} values. The COUNTERS goal $\bigwedge_{i=0}^{n-1}(x_i < x_{i+1})$ results in optimal $h^{FO}_{\max}$ and h^{FO}_{FF} values, as the goal becomes satisfiable only at layer P_n.

From a computational standpoint, computing $h^{FO}_{\max}$ and h^{FO}_{FF} is NP-hard. It is possible to reduce any SAT problem T into a planning problem P such that T is satisfiable iff $h^{FO}_{\max}(s_0) \leq 1$, where s_0 is the initial state of P. For the mapping, we just need Boolean state variables x_i initially set to $\bot$ along with actions a_i that can make each variable x_i true. The goal P is the CNF formula T with the literals p_i and $\neg p_i$ replaced by the atoms x_i and $\neg x_i$ respectively. In practice, however, the formulas appearing in action preconditions and effects, and even in goal formulas, are often simple enough so that the heuristics can be computed efficiently. When that is not the case, the heuristics can be approximated by applying local consistency techniques, trading off informativeness for complexity, and bringing the computation of the FOL-RPG back to polynomial time. We discuss this option in Section 3.6.

3.5 Computation of the First-Order Relaxed Planning Graph

From an operational point of view, the computation of the first-order RPG follows the same philosophy than that of the standard RPG. An outline of the algorithm is given in Fig. 3.4. The computation starts on propositional layer P_0, where the set x^0 of reachable values for each state variable $x \in \mathcal{V}(P)$ contains the single value x^s, i.e. the value of the variable on the state s from which the FOL-RPG is built. Sets x^k in successive layers k of the graph contain by definition all those values in x^{k-1}, plus values newly supported by the (possibly conditional) effect of some ground action that is found to be applicable in the previous propositional layer P_{k-1}. The construction of the FOL-RPG continues until the goal of the problem is satisfied in some layer, or a fixed point is reached. In the first case, a relaxed plan can be extracted from the graph; in the second no relaxed plan exists.

The key operation in the construction of the FOL-RPG is that of checking whether a first-order interpretation exists in the set I_k for a certain layer of the graph that satisfies a given formula. This operation is used at two points in the algorithm:

1. For each problem action $o \in A$ and effect $e \in \text{effs}(o)$, the algorithm iterates through all models $\mathcal{M} \in \mathcal{I}_{k-1}$ from the previous layer P_{k-1} that satisfy both the precondition of the action and the condition of the effect (Fig. 3.4, line 1). For each such model $\mathcal{M}$, the effect is said to *support* the value $[rhs(e)]^{\mathcal{M}}$ for the state variable $var(lhs(e), \mathcal{M})$, where

   - $rhs(e)$ is the right-hand side term of the effect (or the values $\top$ or $\bot$, for relational effects).
   - $lhs(e)$ is the left-hand side term or atom of the effect, which is by definition of the form $f(\bar{t})$ or $R(\bar{t})$, with f and R being *fluent* function and predicate symbols, and $\bar{t} = t_1, \ldots, t_m$ terms over the language.
   - $var(p(\bar{t}), \mathcal{M})$ is the actual state variable, functional or relational, that the symbol p resolves to when applied to the objects denoted by $\bar{t}$ under the model $\mathcal{M}$, i.e. $var(p(\bar{t}), \mathcal{M}) \equiv \underline{p(\bar{t}^{\mathcal{M}})}$.

Algorithm: FOL-RPG(FSTRIPS problem P, state s)

$x^0 := \{x^s\}$ for all $x \in \mathcal{V}(P)$
for $k = 1, 2, \ldots$ **do**
  $x^k := x^{k-1}$ for all $x \in \mathcal{V}(P)$
  forall $o \in A$ *and* $e \in \text{effs}(o)$ **do**
(1) **for** $\mathcal{M} \in models(pre(o) \wedge cond(e), P_{k-1})$ **do**
    $x := var(lhs(e), \mathcal{M})$
    $v := [rhs(e)]^{\mathcal{M}}$
    $x^k := x^k \cup \{v\}$
  $P_k := \{x^k \mid x \in \mathcal{V}(P)\}$
  if $P_k = P_{k-1}$ **then return** $\perp$ `// Fixpoint reached: no solution`
(2) **if** $models(\phi_G, P_k)$ *is not empty* **then**
    Let σ be the value assignment corresponding to any $\mathcal{M} \in models(\phi_G, P_k)$
    Let α be the set of atoms "$x = v$" from σ that support ϕ_G
    return extract(α, $\langle P_0, \ldots, P_k \rangle$)

Figure 3.4: Computation of the First-Order Relaxed Planning Graph. Successive propositional layers P_k are built for $k = 0, 1, \ldots$, each containing a set x^k of reachable values for each state variable x. Values in x^k grow monotonically with k, including all those values in x^{k-1} plus values newly supported by the (possibly conditional) effect of some ground action $o \in A$ that is found to be applicable in the previous propositional layer P_{k-1}.

2. Checking whether there is any model $\mathcal{M} \in \mathcal{I}_k$ in layer P_k of the FOL-RPG that satisfies the goal formula ϕ_G (Fig. 3.4, line 2).

We next show that both operations can be mapped into a standard constraint satisfaction problem that depends both on the formula ϕ whose reachability we want to check and on the particular layer P_k of the relaxed planning graph where we check this reachability. We denote by $\Gamma(\phi, P_k)$ this CSP, which we define next. In general, the intuition is that each solution to $\Gamma(\phi, P_k)$ will correspond to a logical interpretation that satisfies ϕ in layer P_k.

Because it is significantly simpler and is actually sufficient for most of the problems we will be looking at, however, we first present a version of $\Gamma(\phi, P_k)$ that works for the syntactic fragment of FSTRIPS which we call FSTRIPS_0. We use $\Gamma_0(\phi, P_k)$ to denote the simpler constraint satisfaction problem that is sufficient for this fragment. In Section 3.5.3, we provide the full definition of $\Gamma(\phi, P_k)$.

3.5.1 The Functional STRIPS Fragment $\mathsf{FSTRIPS}_0$

The computation of the FOL-RPG is significantly simpler for a syntactic fragment of Functional STRIPS that is nevertheless expressive enough to subsume most of the standard propositional benchmarks from the International Planning Competitions.

Definition 3.7 ($\mathsf{FSTRIPS}_0$)**.** *$\mathsf{FSTRIPS}_0$ is the syntactic fragment of Functional STRIPS where*

1. *Formulas can only be* conjunction of literals*, and*
2. *fluent predicate and function symbols do not appear nested, i.e. in the abstract syntax tree of any formula or term appearing in the problem description, there is no node labeled with a fluent symbol which is the root of a subtree containing some other fluent symbol.*

Without loss of generality, when discussing $\mathsf{FSTRIPS}_0$ we will assume that in all terms $f(\bar{t})$ where f is a fluent symbol and atoms $R(\bar{t})$ where R is a fluent predicate symbol, $\bar{t}$ is a tuple of *constant symbols*, and hence $\underline{f(\bar{t})}$ and $\underline{R(\bar{t})}$ stand for state variables of the problem. Indeed, if that is not the case, then $\bar{t}$ must be a tuple of *fixed* compound terms t_i, which can be replaced at preprocessing by constant symbols c_i such that $t_i^* = c_i^*$ thanks to the restrictions in $\mathsf{FSTRIPS}_0$.

3.5.2 CSP Model for $\mathsf{FSTRIPS}_0$ Formulas

Definition 3.8 ($\Gamma_0(\phi, P_k)$)**.** *Let P be a FSTRIPS problem P over language $\mathcal{L}(P)$, ϕ a variable-free conjunction of literals over the language, and P_k a layer of a first-order relaxed planning graph for the problem, i.e. containing one domain x^k for each state variable $x \in \mathcal{V}(P)$ of the problem. The constraint satisfaction problem $\Gamma_0(\phi, P_k) = \langle X, D, C\rangle$ that has set of variables X, domains D and constraints C is defined by induction over the structure of the formula ϕ. For each term, atom and subformula that appears in ϕ, a number of variables and constraints are added to the CSP, as follows.*

Terms *For each term t in ϕ, X contains one CSP variable v_t, intended to capture the denotation of t under the interpretation that will be derived from each solution to the CSP. The domain $D(v_t)$ of v_t is given by the logical type $type(t)$ of the term, and contains all objects of its (finite) universe $\mathcal{U}_{type(t)}$.*[7]

When t is one of the functional state variables $x \in \mathcal{V}(P)$ of the problem, C contains a domain constraint $v_t \in x^k$, so that v_t can take only those values allowed in P_k. If t is not a state variable, but is a fixed *constant term, then C contains a constraint $v_t = t^*$ that restricts the value of v_t to the only actual value allowed by the planning problem for t.*

Finally, if t is a term of the form $f(t_1, \ldots, t_m)$ where f if a fixed *function symbol, then C contains an extensional ("table") constraint of arity $m+1$ with scope $\langle v_{t_1}, \ldots, v_{t_m}, v_t\rangle$. The constraint contains one tuple $\langle c_1, \ldots, c_m, c\rangle$ for every tuple $\langle c_1, \ldots, c_m\rangle$ in the function domain, and $c = f^*(c_1, \ldots, c_m)$. Intuitively, this constraint ensures that the values taken by the variables $v_{t_1}, \ldots, v_{t_m}$ and v_t are consistent with the fixed denotation of the function f.*

[7] Appendix A provides more details on these first-order logic constructs.

For some fixed function symbols for which the underlying CSP solver accepts a built-in equivalent constraint, such as the arithmetic functions "+", "−", etc., the built-in instead of the table constraint is used. Since nested fluent symbols are not allowed, the above case-based definition covers all possible terms allowed by the FSTRIPS fragment we are discussing.

Atoms *The case of atoms is analogous to that of terms. For each atom p in ϕ, X contains one CSP variable v_p intended to capture the truth value of p under the logical interpretation derived from each solution to the CSP. The domain of v_p is $D(v_p) = \{0, 1\}$.*

If p is one of the relational state variables $x \in \mathcal{V}(P)$ of the problem, C contains a unary domain constraint $v_p \in x^k$. Otherwise, if p is a fixed *constant nullary atom, then C contains a constraint $v_p = p^*$. If p is an atom of the form $R(t_1, \ldots, t_m)$ where R if a* fixed *predicate symbol, then C contains an extensional constraint of arity $m + 1$ with scope $\langle v_{t_1}, \ldots, v_{t_m}, v_t \rangle$. The constraint contains one tuple $\langle c_1, \ldots, c_m, c \rangle$ for every tuple $\langle c_1, \ldots, c_m \rangle$ in the predicate domain, where $c = 1$ if $\langle c_1, \ldots, c_m \rangle \in R^*$, and $c = 0$ otherwise.*

For some fixed relation symbols for which the underlying CSP solver accepts a built-in equivalent constraint, such as the integer comparison operators "=", "≠", "<", "≤", ">", "≥", the built-in rather than the extensional constraint is used, reified *into the Boolean variable v_p.*[8]

Negation and Conjunction of Atoms *For any conjunction of literals $L_1 \wedge \ldots \wedge L_m$, where L_i is an atom p or its negation $\neg p$, a constraint $v_p = 1$ ($v_p = 0$, respectively) is added to C for every positive (negative) occurrence of atom p in the conjunction.*

In some situations where the formula is simple enough (e.g. a conjunction $x < y \wedge y < z$, where x, y and z are fluent logical constants, i.e. state variables), the CSP is optimized and no actual CSP variables are created for the atoms themselves. Instead, non-reified constraints $v_x < v_y$ and $v_y < v_z$ are directly created.

From any solution to the CSP $\Gamma_0(\phi, P_k)$, a (possibly partial) assignment σ to the state variables of the problem P that appear in ϕ can be directly built. Let us denote by $\mathcal{I}(\sigma)$ the set of first-order interpretations for $\mathcal{L}(P)$ that contains all interpretations $\mathcal{M}$ consistent with σ, i.e. such that for every state variable x appearing in ϕ, $x^{\mathcal{M}} = \sigma(x)$. Because of the construction of $\Gamma_0(\phi, P_k)$, we have that $\mathcal{I}(\sigma)$ contains exactly all those interpretations in $\mathcal{I}_k$ that satisfy ϕ.

Example 3.9. *Assume we have a FSTRIPS* COUNTERS *instance (see Fig. 3.3) with* 3 *variables x_1, x_2, x_3 taking values within the interval $[0, 5]$. As discussed before, layer P_1 of the FOL-RPG is made up of state variable domains $x_1^1 = x_2^1 = x_3^1 = \{0, 1\}$. In order to check whether the goal formula $\phi_G \equiv x_1 < x_2 \wedge x_2 < x_3$ is satisfied in layer P_1, the set of first-order interpretations $\mathcal{I}_1$ (i.e. the set of interpretations consistent with the state variable domains in P_1) is explored to find some model of ϕ_G. To do this, a solution of the constraint satisfaction problem $\Gamma_0(\phi_G, P_1)$ is sought.* $\Gamma_0(\phi_G, P_1)$

[8] In constraint programming, a constraint c is said to be *reified* into a Boolean "control" variable b if the truth value of the constraint under any given assignment of values to variables is bound to b, i.e. $b = 1$ iff the constraint holds (Smith, 2006; Feydy et al., 2011).

has integer variables v_{x_1}, v_{x_2}, v_{x_3} with domains $D(v_{x_1}) = D(v_{x_2}) = D(v_{x_3}) = \{0, 1\}$; each of these variables models, respectively, the value taken by terms x_1, x_2 and x_3 of the formula ϕ_G. The two goal atoms $x_1 < x_2$ and $x_2 < x_3$ are mapped into CSP arithmetic constraints $v_{x_1} < v_{x_2}$ and $v_{x_2} < v_{x_3}$. It can be seen that the CSP has no solutions, from which we conclude that ϕ_G is not reachable in P_1 On the contrary, layer P_2 has state variable domains $x_1^2 = x_2^2 = x_3^2 = \{0, 1, 2\}$, and the constraint satisfaction problem $\Gamma_0(\phi_G, P_2)$ is defined as above, but with CSP domains $D(v_{x_1}) = D(v_{x_2}) = D(v_{x_3}) = \{0, 1, 2\}$. The assignment $v_{x_1} = 0$, $v_{x_2} = 1$ and $v_{x_3} = 2$ is a solution to the CSP, hence ϕ_G is satisfiable, and that assignment is the basis from which the relaxed plan extraction proceeds.

3.5.3 CSP Model for Arbitrary FSTRIPS Formulas

The above syntactic restrictions on the class of FSTRIPS problems that can be addressed with $\Gamma_0(\phi, P_k)$ can be dropped with some extra work to account for (a) the possibility of using arbitrary logical connectives other than conjunctions, and (b) the possibility of having nested fluents. We formalize this below.

Allowing arbitrary connectives is rather straight-forward, once atoms p are reified into Boolean variables v_p as described above. This is common practice in constraint programming modeling languages such as Essence (Frisch et al., 2007) or Zinc (Marriott et al., 2008), which transform expressive models into existentially quantified conjunctions of primitive constraints, of the form $\exists \bar{x} \bigwedge_{c \in C} c$, through *reification* and *flattening* (Feydy et al., 2011). A formula such as $(x < y \vee x < z) \wedge (x > w)$, for instance, can be translated with the help of 4 extra Boolean variables $b_1, \ldots, b_4$, into a set of 4 reified primitive constraints

- $x < y \iff b_1$,
- $x < z \iff b_2$,
- $x > w \iff b_3$,
- $(b_1 \vee b_2) \iff b_4$

plus one final constraint $b_4 \wedge b_3$ (Marriott et al., 2008).

Dealing with arbitrarily nested fluents is slightly more involved. To illustrate, consider the unsatisfiable formula ϕ:

$$f(0) = 0 \wedge f(f(0)) = 1$$

involving a fluent function symbol f defined over domain and range $\{0, 1\}$. Assume that $f(0)$ and $f(1)$ are the only two state variables of the problem, and the RPG layer P_k dictates reachable values $\{0, 1\}$ for both of them. In the constraint satisfaction problem $\Gamma_0(\phi, P_k)$ as defined above, the denotation of terms $f(0)$ and $f(f(0))$ is modeled by CSP variables $v_{f(0)}$ and $v_{f(f(0))}$, respectively, and the set of constraints of the problem is:

- $v_{f(0)} \in \{0, 1\}$
- $v_{f(f(0))} \in \{0, 1\}$
- $v_{f(0)} = 0$
- $v_{f(f(0))} = 1$

The assignment $v_{f(0)} = 0$, $v_{f(f(0))} = 1$ is a solution of the CSP, since there is no constraint binding the values of variables $v_{f(0)}$ and $v_{f(f(0))}$. By definition, however, f is a function symbol, and its denotation under any interpretation must be an actual function, meaning that if $f(0) = 0$, then $f(f(0)) = f(0) = 0$ as well.

We can enforce that the values of different CSP variables potentially denoting the same function point be consistent by making use of an *element* constraint. In constraint satisfaction, an element constraint (Van Hentenryck and Carillon, 1988; van Hoeve and Katriel, 2006) binds the value taken by an *index* integer variable $I \in [0, m]$, the values taken by an *array* T of $m + 1$ integer variables $T[0]$, $T[1]$, ..., $T[m]$, and a *result* variable integer R, so that the index variable does indeed act as an index to the array of variables, and it holds that $T[I] = R$.

The general formulation of the CSP $\Gamma(\phi, P_k)$ takes both aspects into account, and essentially extends the definition of $\Gamma_0(\phi, P_k)$ with some additional constraints. We consider here quantification-free problems only, and deal with existential quantification in Chapter 4.

Definition 3.10 ($\Gamma(\phi, P_k)$)**.** *Let P be a FSTRIPS problem P over language $\mathcal{L}(P)$, ϕ an arbitrary variable-free formula of the language and P_k a layer of a first-order relaxed planning graph for the problem, i.e. containing one domain x^k for each state variable $x \in \mathcal{V}(P)$ of the problem. Let $\Gamma_0(\phi, P_k)$ be the constraint satisfaction problem as defined in Definition 3.8. The constraint satisfaction problem $\Gamma(\phi, P_k) = \langle X, D, C \rangle$ contains all variables, corresponding domains and constraints specified in $\Gamma_0(\phi, P_k)$, plus some additional elements for each term, atom and formula appearing in ϕ, which are the following:*

Terms *If t is a term of the form $f(t_1, \ldots, t_m)$, where f is a* fluent *function symbol and at least one of the terms t_1, ..., t_m contains some occurrence of a* nested *fluent symbol, then C contains an* element *constraint constraining the value taken by v_t to be consistent with the values taken by variables $v_{t_1}, \ldots, v_{t_m}$ and by any other variable $v_{t'}$ in the CSP where t' is a* state variable that results from the same fluent symbol f*. To do so, every possible combination of values for terms $t_1, \ldots, t_m$ is given a* unique index *with an extra integer variable I and an extra extensional constraint with scope $\langle v_{t_1}, \ldots, v_{t_m}, I \rangle$. The element constraint then has index variable I and result variable v_t, and the array of variables contains the (ordered) CSP variables $v_{\underline{x_1}}, v_{\underline{x_2}}, \ldots, v_{\underline{x_l}}$, where $\underline{x_i}$ is the state variable formed by symbol f plus arguments given by the tuple in the function domain that corresponds to index value $I = i$.*

Atoms *If ϕ contains some atom $R(t_1, \ldots, t_m)$, with R a* fluent *predicate symbol and some other nested (function) symbol, then C contains extra variables and constraints exactly in the same manner as in the case of function symbols above.*

Negation, Conjunction and Disjunction of Formulas *Let us assume without loss of generality that the (variable-free) ϕ is in negation normal form, i.e. it is constructed from literals using $\wedge$ and $\vee$. For each literal L in ϕ which is the negation $\neg p$ of some atom p, X contains a Boolean CSP variable v_L, and C contains the reified constraint $\neg v_p \iff v_L$. For each subformula φ of the form $L_1 \otimes L_2$ in ϕ, where $\otimes \in \{\wedge, \vee\}$, X contains a Boolean CSP variable v_φ, and C contains the reified constraint $(v_{L_1} \otimes v_{L_2}) \iff v_\varphi$. Finally, C contains*

a top-level constraint $v_\phi = 1$. As described before, in some situations where the formula is simple enough, the CSP is optimized and simpler, non-reified constraints are used.

3.6 Approximation of the First-Order Relaxed Planning Graph

We look now at methods for approximating the first-order relaxed planning graph in polynomial time, in order to derive heuristics that are more informed than those resulting from the standard RPG, but remain computationally tractable. The intractability of the FOL-RPG follows from checking whether there is an interpretation $\mathcal{M}$ in a potentially exponentially-sized set of interpretations I_k valid in a certain layer P_k of the RPG that satisfies a certain formula. In Sections 3.5.2 and 3.5.3 above, we have seen how this can be mapped into a constraint satisfaction problem with size linear on the size of the planning problem.

The CSP that models the satisfiability of the formula $pre(a) \wedge \phi \wedge w = v$ for each action a and conditional effect $\phi \rightarrow f(t) := w$, which is posted to determine whether a supports the value v for a certain state variable, usually involves a bounded and small set of state variables, and can thus be fully solved. On the other hand, the CSP that represents models the satisfiability of the goal formula ϕ_G might be in some cases too hard to solve when ϕ_G is a complex formula. Note, however, that for most of the standard existing classical planning benchmarks, which can be expressed in the $\mathsf{FSTRIPS}_0$ fragment, this formula will be a conjunction of fluent atoms of the form $p_1 \wedge \ldots \wedge p_n$, and the corresponding CSP will be solved in linear time. For complex FSTRIPS formulas, the corresponding CSP can be solved *approximately* by using various forms of *local consistency*. In other words, the approximation in the construction of the FOL-RPG will typically apply only to checking whether the goal ϕ_G is satisfiable in a propositional layer P_k, and relies on the notions of polynomial arc and node consistency that were discussed in Section 2.6. In our case, the goal ϕ_G is *approximated as being satisfiable* in some RPG layer P_k if the corresponding CSP $\Gamma(\phi_G, P_k)$ is locally consistent, i.e. if after propagating all constraints and pruning the domains of the CSP variables, no variable gets an empty domain. The $h^{apx}_{\max}$ heuristic is defined by the index k of the first layer P_k where the goal is satisfiable, and it is a polynomial approximation of the intractable $h^{FO}_{\max}$ heuristic. It is easy to see that $0 \leq h_{\max} \leq h^{apx}_{\max} \leq h^{FO}_{\max} \leq h^*$, where h^* is the optimal heuristic. The pruning resulting from the goal CSP is used also in the *plan extraction procedure* that underlies the computation of the h^{apx}_{FF} heuristic, where pruned values of goal variables are excluded. As an example, if the goal is the atom $x = y$ where x and y are state variables with domains $D(x) = \{e, d\}$ and $D(y) = \{d, f\}$, arc consistency will prune the value e from $D(x)$ and f from $D(y)$, so that plan extraction will backchain from atoms $x = d$ and $y = d$.

3.7 Language Extensions

The computation of the FOL-RPG through a mapping into a constraint satisfaction problem opens up a number of interesting extensions to the language, which we review next.

3.7.1 Global Constraints

In constraint satisfaction, global constraints are constraints with arbitrary but usually large arity that offer a compact way to model some important concept that arises frequently in a number of problems (Bessiere and Van Hentenryck, 2003; van Hoeve and Katriel, 2006). As we discussed in Section 2.6, global constraints are useful not only from an expressive point of view, but also from a computational one. The classical example to illustrate this is the global constraint $\texttt{alldiff}(x_1, \ldots, x_n)$, which declares that the values of the n variables x_i need to be different, and can be seen as a compact way of encoding a quadratic number of binary constraints $x_i \neq x_j$. The constraint $\texttt{alldiff}(x_1, \ldots, x_n)$ is not only more compact, but has stronger propagation. Indeed, arc consistency over the set of $x_i \neq x_j$ constraints finds no inconsistency when the domains are e.g. $D(x_i) = \{0, 1\}$ for all i, yet it is known that such constraints cannot be jointly satisfied if $|\cup_{i=1}^{n} D(x_i)| < n$.

Global constraints can be a useful addition to planning as well. As a simple illustration, the goal of stacking all blocks in a single tower in BLOCKSWORLD, regardless of their relative positions, can be compactly encoded with the global constraint $\texttt{alldiff}(loc(b_1), \ldots, loc(b_n))$. Even standard classical planning benchmarks often have features that could be best captured with global constraints. The CHILDSNACK domain from the 2014 International Planning Competition, for instance, requires that a number of children be served each one a different sandwich that satisfies certain dietary restrictions. A possible FSTRIPS reformulation can feature an atom $\texttt{alldiff}(served(c_1), \ldots, served(c_n))$ in the goal conjunction in order to encode that requirement in a compact manner. Other global constraints such as arithmetic constraints, or more application-oriented *packing* and *scheduling* constraints (Beldiceanu et al., 2012), also possess an immediate relevance for planning. We discuss further examples involving the use of global constraints as state constraints in Section 3.7.3.

In our context, a global constraint can simply be seen as nothing else than a convenient manner of providing the denotation of a *fixed* predicate symbol of the first-order language $\mathcal{L}(P)$ over which a FSTRIPS problem P is defined. Thus, they are dealt in the same manner than e.g. a predicate symbol "$<$": during the computation of the FOL-RPG, the built-in global constraints of the underlying CSP solver are used instead of extensional constraints; whereas during the search, the CSP solver can be invoked in order to provide the actual denotation of the symbol or, if performance is crucial, simple ad-hoc procedures can be implemented for each particular global constraint to provide that denotation without having to perform a costly invocation of a CSP solver. When the complexity of the formula makes it necessary for the computation of the FOL-RPG to reify the CSP constraints corresponding to the atoms in the formula (see Sections 3.5.2 and 3.5.3), some global constraints might not be available, since not all CSP solvers offer reified versions of all global constraints (Feydy et al., 2011). This means, for instance, that a disjunction of different `alldiff` atoms might prevent the computation of the FOL-RPG from using off-the-shelf solvers.

3.7.2 Externally-Defined Symbols

As we just saw, global constraints are no other thing that a convenient way of specifying the denotation of *fixed* symbols of the language, but they are not the only convenient way of doing so. Indeed, as long as some mechanism is provided that supports the operations that are necessary during the search and computation of the heuristic, any fixed symbol R can be denoted by any external procedure. In

our case, these operations essentially are (a) providing the denotation of the symbol on any point in the symbol domain, and (b) for the computation of the FOL-RPG, providing a (polynomial) constraint propagator that is able to enforce some form of local consistency over domains $D(v_i)$ for any expression of the form $R(v_1, \ldots, v_m)$, where v_i are CSP variables with domains $D(v_i)$. This latter requirement is far from trivial, but when the arity of the symbol and the involved domains are small, an *extensional* constraint can efficiently be used to fulfill it, and can always be computed for any arbitrary denotation. When no heuristics are required, as in Chapter 5, this requirement can be dropped.

Externally-defined symbols are extremely useful to model aspects which are not easy to capture in a declarative way, such as e.g. geometrical constraints on packing problems. In Chapter 5, we will use them to model things such as the (deterministic) motion of ghosts in PACMAN, or the billiard-like ball motion of PONG. The possibility of giving the (fixed) denotation of some of the logical symbols of the language by means of *external procedures* implemented in some programming language was first considered theoretically in (Geffner, 2000), and given support in the `GPT` planner (Bonet and Geffner, 2001a). The practicality of the notion led to its revitalization several years after, this time under the name of *semantic attachments* (Dornhege et al., 2009, 2012; Hertle et al., 2012), and to efforts to give adequate heuristic support to problems with this type of feature (Bernardini et al., 2017).

3.7.3 State Constraints

Another relatively straight-forward extension of the language allowed by the FOL-RPG model is that of *state constraints*, which are arbitrary formulas that must be true in all states encountered through the execution of a plan. The convenience of state constraints has been thoroughly discussed in the literature on reasoning about actions (Lin and Reiter, 1994; Son et al., 2005), sometimes under the name of *plan* or *state invariants*. State constraints however should not be confused with state invariants: a state constraint *enforces* a formula to be true in all reachable states, while a state invariant is a formula that can be proven to hold in all reachable states. For example, block a not being on top of two blocks at the same time is a typical state invariant in blocksworld, whereas a not being on top of block b might be a particular state constraint that we want to enforce. State constraints can thus be thought of as a convenient way to express and make use of information about:

1. Preconditions that are implicit for all actions: actions leading to the violation of a state constraint are deemed as not applicable.
2. States to be avoided, e.g., states where a formula $p \wedge q$ is true. State constraints ϕ used this way model the class of extended temporal goals "never ϕ" (Dal Lago et al., 2002).
3. Dead-end conditions, that is, conditions that if ever achieved preclude reaching the goal.

Actually, a state constraint ϕ can also be seen as the particular Linear Temporal Logic (LTL) formula "always ϕ" (Huth and Ryan, 2004), and as such is subsumed by the *state trajectory constraints* in the PDDL 3.0 specification (Gerevini and Long, 2005), which in spite of offering interesting expressive power, have received little support from existing planners. Indeed, state constraints are also related to *safety*

constraints (Weld and Etzioni, 1994), which were already present in the original PDDL formalism (McDermott et al., 1998, Section 10).

State constraints can be convenient for both modeling *and* computation. The classical Missionaries and Cannibals problem (Amarel, 1968; McCarthy, 1998), for instance, has 3 missionaries and 3 cannibals on the left bank of a river which they all want to cross. A single boat is available that can take at most 2 people at a time, missionaries or cannibals alike. Missionaries do not want to be outnumbered by the cannibals, be it on either bank of the river or inside of the boat, for fear of the cannibals exercising their defining inclination.[9] The goal is to find an appropriate schedule of river crossings that transports everyone to the right bank of the river in a safe manner. A generalization of the problem for n missionaries and n cannibals, $n \geq 3$, on a complete graph, can be easily modeled in FSTRIPS with the help of state constraints. For this, there is a set of locations $l_1, \ldots, l_m$, and terms $nc(l)$ and $nm(l)$ that represent the number of cannibals and missionaries at each location l. The nullary function symbol x represents the current location of the boat. One single action schema move(c, m, x') is needed, with precondition

$$(1 \leq c + m \leq 2) \wedge (c \leq nc(x)) \wedge (m \leq nm(x))$$

which moves c cannibals and m missionaries in one boat trip from the current location x to x', thanks to effects $x := x'$, $nc(x') := nc(x') + c$, $nc(x) := nc(x) - c$, $nm(x') := nm(x') + m$, $nm(x) := nm(x) - m$. Most relevantly, the restriction on cannibals not outnumbering missionaries is modeled by a state constraint $\bigwedge_l (nm(l) \geq nc(l) \vee nm(l) = 0)$.

As a different example, in (Ferrer-Mestres et al., 2015) we show how state constraints can be conveniently used to express *non-overlap* conditions between the different objects of task and motion planning problems. Another example, the well-known Sokoban domain can also be encoded more compactly and elegantly by making use of state constraints. Figure 3.5 shows a possible encoding, where the location of the agent and of the stones (or boxes) is encoded through a *loc* fluent function symbol, and the geometry of the 2-dimensional grid through a *next* fixed function symbol, such that $next(l, d)$ denotes the grid location which is adjacent to l when moving in direction d. A single state constraint $\texttt{alldiff}(loc(p), loc(s_1), \ldots, loc(s_m))$ is used to ensure that no two things, stones or agent alike, are on the same grid location at the same time. This allows us to get rid of the traditional *clear* predicate, whose (redundant) denotation needs to be updated on every action, and allows the heuristics based on the relaxed planning graph to be more accurate, as we show in the Results section.

We next discuss the minor changes required in the language and in the derivation of heuristics for accommodating state constraints.

Syntax and Semantics. A FSTRIPS problem p with *state constraints* is a standard FSTRIPS problem with the addition of a first-order formula ϕ_C over the language $\mathcal{L}(P)$, which is used to encode *implicit preconditions*. The semantics of the problem is thus equal to the one given in Section 2.3.6 for standard FSTRIPS problems, except

[9] A historically more accurate version of the problem has it that it is the cannibals that do not want to be outnumbered by the missionaries for fear of being converted, but we restrict our discussion to the first version for the sake of tradition.

```
types thing, location, direction
      player subtype of thing,
      stone subtype of thing

objects
    l_1, ..., l_n, ⊥: location
    p: player
    s_1, ..., s_m: stone
    N, E, S, W: direction

predicate next(l: location, d: direction)

function loc(t: thing): location

action move(p: player, d: direction)
  prec next(loc(p), d) ≠ ⊥
  effs loc(p) := next(loc(p), d)

action push(p: player, d: direction, s: stone)
  prec next(loc(p), d) = loc(s) ∧ next(loc(s), d) ≠ ⊥
  effs loc(p) := loc(s)
       loc(s) := next(loc(s), d)

state-constraint alldiff(loc(p), loc(s_1), ..., loc(s_m))

init loc(p) = l_2, loc(s_1) = l_15, ..., loc(s_m) = l_24,
     next(l_1, N) = l_2, ...

goal loc(s_1) = l_7 ∧ loc(s_2) = l_9 ∧ ...
```

Figure 3.5: Functional STRIPS encoding of a classical SOKOBAN instance (simplified syntax), where an agent needs to move a number of boxes or stones in a grid-like warehouse in order to place them in certain *goal* locations, the challenge arising from the fact that stones can be pushed, but not pulled. The encoding features action schemas *move* and *push*, which move the agent alone and pushing a stone, respectively. The use of state constraints makes unnecessary the traditional *clear* predicate.

that the transition function f of the corresponding classical planning model $\mathcal{S}(P)$ is not defined on any state that violates the state constraint formula, i.e. does not contain any transition $\langle s, o, s' \rangle \in S \times O \times S$ such that $\mathcal{M}(s) \models \neg\phi_C$ or $\mathcal{M}(s') \models \neg\phi_C$. As a result, if $s_0, \ldots, s_n$ is the sequence of states generated by a plan that solves P, then ϕ_C will be true in all the states s_i, $i = 0, \ldots, n$.

Heuristics. The introduction of state constraints affects the definition and computation of the first-order RPG and of the derived heuristics $h^{FO}_{\max}$ and h^{FO}_{FF} and their polynomial approximations $h^{apx}_{\max}$ and h^{apx}_{FF}. The changes, however, are minor: In the presence of state constraints, the set of first-order interpretations $\mathcal{I}_k$ that is considered for satisfiability purposes in each propositional layer P_k of the first-order RPG is not allowed to contain any interpretation that does not satisfy the state constraint formula ϕ_C. In the polynomial approximation that leads to the heuristics $h^{apx}_{\max}$ and h^{apx}_{FF}, the state constraints are not used to prune interpretations directly, but just the domains x^k of the state variables x in layer k through constraint propagation.

3.8 Empirical Evaluation

3.8.1 Setup

The results reported in this section were first published in (Francès and Geffner, 2015), using a preliminary version of the `FS` planner which was able to deal with the $\mathsf{FSTRIPS}_0$ fragment of the Functional STRIPS language extended with externally-defined symbols and support for the `alldiff` and `sum` global constraints. In this preliminary version, the constraint satisfaction problems $\Gamma_0(\phi, P_k)$ arising in the computation of the FOL-RPG were solved by using handcoded algorithms and local consistency propagation routines. The planner has since evolved to interface with the `Gecode` constraint solver (Gecode Team, 2006); a full account of the current capabilities of the planner can be found in Chapter 6. Here we report results on the use of a greedy best-first search (GBFS) strategy guided by the h^{apx}_{FF} heuristic. Full details of the benchmarks and the exact version of the planner source code used to run these experiments can be found online at `www.gfrances.github.io`.

We run the `FS` planner on a number of domains, described below, to compare its results with those obtained by some standard planners on equivalent PDDL models, up to language limitations. In order to understand the differences in accuracy and computational cost between the h_{FF} and h^{apx}_{FF} heuristics, we run the `FF` and `Metric-FF` (Hoffmann and Nebel, 2001a; Hoffmann, 2003) planners (the latter on encodings with numeric fluents, when suitable), using the same greedy best-first search strategy (i.e. $f(n) = h(n)$ and enforced hill-climbing disabled). To complete the picture, we also run the state-of-the-art `Fast-Downward` planner with the `LAMA-2011` configuration, which uses different search algorithms and exploits additional heuristic information derived from helpful actions and landmarks (Helmert, 2006a; Richter and Westphal, 2010). `Metric-FF` and `LAMA` results are not shown in the tables but discussed on the text — this is because for `Metric-FF` there is too large a gap in coverage, and in the case of `LAMA`, the different heuristics and search algorithms do not allow for a meaningful comparison other than in terms of coverage. All planners are run a maximum of 30 minutes on a cluster with AMD Opteron 6300@2.4Ghz nodes, and are allowed a maximum of 8GB of memory. We focus on coverage, plan length, number of expanded nodes and total runtime of the plan search. Table 3.1

shows summary statistics for all domains, whereas Table 3.2 shows detailed results for selected instances from each of the domains.

3.8.2 Results

We next describe and discuss, domain by domain, the empirical results of the planner.

Counters

As discussed above, the COUNTERS domains features a number of positive integer variables $x_1, \ldots, x_n$ that take values in some interval $[0, m]$, and the goal is to reach the inequalities $x_i < x_{i+1}$, for $i \in \{0, \ldots, n-1\}$, by applying actions that increase or decrease by one the value of a single variable. We consider three variations of the problem that differ in the way in which variables are initialized:

1. All variables initialized to zero (COUNT_0).
2. All variables initialized to random values in the $[0, 2n]$ range ($\text{COUNT}_{\text{RND}}$).
3. All variables initialized to decreasing values from the $[0, 2n]$ range ($\text{COUNT}_{\text{INV}}$).

The results in Tables 3.1 and 3.2 reveal that for both `FF` and `FS`, the number of expanded nodes agrees with plan length, meaning that plans are found greedily. Plans found by the `FS` planner, however, are consistently shorter. Incidentally, the relaxed plans computed with the approximate FOL-RPG are valid plans for the non-relaxed problem, which is not the case for the standard RPG used by `FF`.

`FF` performs quite well in COUNT_0, where the improved heuristic accuracy offered by `FS` does not compensate the increased running times. This is no longer true if the values of consecutive variables are decreasing. Both in $\text{COUNT}_{\text{RND}}$ and $\text{COUNT}_{\text{INV}}$, `FF` shows significantly poorer performance, solving only instances up to 20 and 12 variables, respectively. As discussed, this results from the fact that each of the $x_i < x_j$ inequalities is conceived as *independent*, since they are actually encoded in apparently independent propositional atoms $less(i, i+1)$, which frequently guides the search towards heuristic plateaus. This is not the case for the `FS` planner, which in spite of a much slower node expansion rate, has better coverage on these two variations of the problem, finding much shorter plans and expanding a consistently smaller amount of nodes.

Grouping

Figure 3.6 shows a fragment of what we call the GROUPING domain, in which some blocks of different colors are scattered on a grid, and we want to group them so that two blocks are in the same cell iff they have the same color. In standard PDDL, there is no compact manner of modeling a goal atom such as $loc(b_1) = loc(b_2)$, unless existentially quantified variables are used. Since that type of quantification is not well-supported by current planners (we will have more to say about that in Chapter 4), we have devised an alternative PDDL formulation with two additional actions which are used, respectively, to (1) *tag* any cell as the *destination cell* for all blocks of a certain color, and (2) *secure* a block in its destination cell. We generate random instances with increasing grid size $s \times s$, $s \in \{5, 7, 9\}$, number of blocks $b \in \{5, 10, 15, 20, 30, 35, 40\}$ and number of colors c between 2 and 10, where blocks are assigned random colors and initial locations.

```
types block, cell, color, direction

functions
  loc(b: block): cell
  next(x: cell, d: direction): cell
  color(b: block): color

action move(b: block, d: direction)
  prec @in-grid(next(loc(b), d))
  effs loc(b) := next(loc(b), d)

goal ∀b1, b2 : block (loc(b1) = loc(b2)) ⇔ (color(b1) = color(b2))
```

Figure 3.6: Fragment of a FSTRIPS encoding of the GROUPING domain, in which the blocks must be moved until they occupy the same cell iff they belong to the same group. The "@" in predicate symbol *@in-grid* denotes that its denotation is provided by an external function; @in-grid(x, d) holds whenever the cell reached by moving in direction d from cell x is within the grid. The universally quantified goal predicate gets expanded over the finite (typed) universe of blocks at preprocessing; since *color* is a fixed predicate, the truth values of atoms $color(b_1) = color(b_2)$ can be computed at preprocessing time as well, and hence the resulting formula is a conjunction of literals of the form $loc(b_1) = loc(b_2)$ or $loc(b_1) \neq loc(b_2)$.

Domain	#I	#C	Coverage		Plan length			Node expansions			Time (s.)		
			FF	FS	FF	FS	R	FF	FS	R	FF	FS	R
COUNT-0	13	11	**13**	11	770.09	**270.09**	2.51	770.09	**270.09**	2.51	**33.98**	318.42	0.13
COUNT-I	13	7	8	**9**	946.14	**204.43**	4.99	946.14	**204.43**	4.99	**34.16**	272.11	0.65
COUNT-R	39	17	17	**28**	499.00	**87.35**	4.10	499.00	**88.76**	4.04	154.50	**25.68**	3.39
GROUP.	72	42	48	**55**	424.24	**43.86**	9.61	681.24	**104.79**	12.28	354.45	**83.12**	41.57
GARD.	51	20	20	**33**	366.85	**86.55**	4.10	2635.95	**456.45**	14.28	205.89	**7.84**	39.68
PUSH.	17	5	5	**8**	65.40	**34.20**	1.43	404.60	**64.80**	3.18	**0.07**	3.98	0.01
PUSH-R	81	34	**53**	34	121.29	**59.38**	1.67	2964.88	**624.82**	7.23	**3.38**	214.33	0.03

Table 3.1: **Summary of results** for FF and FS using a greedy best-first search with heuristics h_{FF} and h^{apx}_{FF} (FF's EHC disabled). #I denotes total number of instances and #C number of instances solved by both planners. Length, node expansion and time figures are averages over instances solved by both planners; R (for *ratio*) is the average of the per-instance FF / FS ratios. Best-of-class figures are shown in bold. LAMA and Metric-FF results are discussed in the text.

Instance	Plan length		Node expansions		Time (s.)	
	FF	FS	FF	FS	FF	FS
COUNT$_0$ ($n = 8$)	68	**28**	68	**28**	**0.03**	0.14
COUNT$_0$ ($n = 20$)	530	**190**	530	**190**	**2.46**	25.13
COUNT$_0$ ($n = 40$)	2260	**780**	2260	**780**	**197.65**	1796.9
COUNT$_{\text{INV}}$ ($n = 8$)	54	**48**	54	**48**	**0.06**	0.36
COUNT$_{\text{INV}}$ ($n = 20$)	416	**300**	416	**300**	**8.62**	99.99
COUNT$_{\text{INV}}$ ($n = 44$)	**2148**	-	**2148**	-	**1073.46**	-
COUNT$_{\text{RND}}$ ($n = 8$)	30	**26**	30	**26**	**0.05**	0.1
COUNT$_{\text{RND}}$ ($n = 20$)	2250	**227**	2250	**227**	333.41	**37.1**
COUNT$_{\text{RND}}$ ($n = 36$)	-	**680**	-	**680**	-	**1422.83**
GROUP. ($s = 5, b = 10, c = 2$)	287	**23**	391	**23**	3.34	**0.57**
GROUP. ($s = 7, b = 20, c = 5$)	215	**37**	259	**105**	**9.75**	26.39
GROUP. ($s = 9, b = 30, c = 7$)	638	**98**	752	**98**	1346.79	**169.78**
GARD. ($s = 5, k = 4$)	121	**30**	144	**34**	3.47	**0.12**
GARD. ($s = 10, k = 10$)	1156	**155**	8827	**244**	1058.55	**7.54**
GARD. ($s = 15, k = 22$)	-	**360**	-	**1580**	-	**242.45**
PUSH ($s = 7, k = 4$)	51	**49**	**85**	102	**0.02**	4.27
PUSH ($s = 10, k = 7$)	-	**189**	-	**576**	-	**160.52**
PUSH$_{\text{RND}}$ ($s = 7, k = 4$)	**36**	38	**75**	105	**0.02**	3.16
PUSH$_{\text{RND}}$ ($s = 10, k = 8$)	423	**113**	31018	**4867**	**59.92**	1024.59

Table 3.2: **Details on selected instances** for the FF and FS planners. A dash indicates the solver timed out before finding a plan, and best-of-class numbers are shown in bold typeface. Particular instance parameters are described on the text.

Results show that the coverage of FS is slightly higher, and the h^{apx}_{FF} heuristic used by FS proves to be much more informed than the propositional version used by FF, resulting on average on 12 times less node expansions and plans around 10 times shorter. This is likely because the delete-free relaxation of the problem allows all unpainted cells to be painted of all colors in the first layer of the RPG, thus producing poor heuristic guidance. FS does not incur on this type of distortion, as it understands that goal atoms such as $loc(b_1) = loc(b_2)$ and $loc(b_2) \neq loc(b_3)$ are constrained *to be satisfied at the same time*. Comparing both planners, the increased heuristic accuracy largely compensates in terms of search time the cost of the polynomial approximate solution of the CSP based on local consistency.

Gardening

We now illustrate an additional way in which global constraints can greatly improve not only the modeling process but also the accuracy of the heuristic estimates. In the GARDENING domain, an agent in a grid needs to water several plants with a certain amount of water that is loaded from a tap and poured into the plants unit by unit. It is well-understood that standard delete-free heuristics are misleading in this type of planning-with-resources environments (Coles et al., 2008), since they fail to account for the fact that the agent needs to load water *repeatedly*: in a delete-free world, one unit of water is enough to water all plants. As a consequence, the plans computed following delete-free heuristics tend to have the agent going back and forth to the water tap, loading each time a single unit of water. FS, however, is actually able to accommodate and use a *flow constraint* equating the total amount of water obtained from the tap with the total amount of water poured into the plants. This only requires state variables $poured(p_1), \ldots, poured(p_n)$, and a variable T intended to denote the total amount of obtained water, plus a goal sum constraint $poured(p_1) + \cdots + poured(p_n) = T$.

We generate random instances with grid size $s \times s$, $s \in [4, 20]$, having one single water tap and $k = max(4, \lfloor s^2/10 \rfloor)$ plants to be watered, each with a random amount of water units ranging from 1 to 10. The goal and preconditions of the problem are simple enough not to involve more than once the same state variable, so we are just testing how a reasonably simple goal constraint can improve the heuristic accuracy of the first-order RPG. The results indeed show that FS significantly and systematically outperforms FF in all aspects, offering higher coverage and finding plans 4 times shorter, almost 40 times faster, on average.

Pushing

Finally, the PUSH domain is a simplification of the well-known SOKOBAN domain where obstacles have been lifted. In PUSH, an agent moves around an obstacle-free grid, pushing k stones into k fixed different goal cells. A noticeable inconvenient of delete-free heuristics in this type of domain is that relaxed plans tend to push all stones to the closer goal cell, even if this means that several stones end up on the same cell. This is because the delete-free nature of the relaxation means that if a given cell is *clear* in the state on which the RPG is computed, it remains clear throughout the whole relaxed plan. FSTRIPS allows us to express the (implicit) fact that stones must be placed in different cells through the use of a global constraint alldiff$(loc(s_1), \ldots, loc(s_k))$, which FS exploits to infer more informative heuristic values. This stands in contrast to other heuristic planners, which might be able to

indirectly model such a constraint but not to leverage it to improve the heuristic and the search.

To specifically test the potential impact of this constraint, we model a variation of the problem in which all stones concentrate near a single goal cell, with the rest of goal cells located on the other side of the grid (PUSH). We also test another variation where agent, stones and goal cells are assigned random initial positions (PUSH_{RND}). For the first type of instances, `FS` is indeed able to heuristically exploit the `alldiff` constraint and scale up better, having an overall larger coverage, and finding significantly shorter plans. In random instances, on the other hand, `FF` offers slightly better coverage and a notably smaller runtime, likely because the impact of the above-mentioned heuristic distortion is much smaller in this setting, but in terms of plan length and number of expanded nodes, `FS` still outperforms `FF` by a large margin.

3.8.3 Other Planners and Overview

We now briefly discuss the performance of the `LAMA` and `Metric-FF` planners on the same set of problems. In the COUNTERS domain family, `LAMA` offers a somewhat uneven performance, solving 27/33 instances of the random variation, but only 5/11 of the other two variations. Average plan length is in the three cases significantly better than `FF`, but at least a 30% worse than `FS`, and while total runtimes decidedly dominate the two GBFS planners in the `rnd` and `inv` version, they are much worse in COUNT_0. For `Metric-FF`, the PDDL 2.1 encoding that we use is identical to the FSTRIPS encoding, save minor syntactic differences. `Metric-FF` solves 8/11 instances in COUNT_0, but for the other variations only solves a couple of instances. Given that the model is the same, this strongly suggests that at least in some cases it pays off to properly account for the constraints among state variables involved in several atoms.

In the GROUPING domain, `LAMA` has perfect coverage and is much faster than the two GBFS-based planners, but again the `FS` plans are significantly shorter. In the GARDENING domain, `LAMA` performs notably worse than `FS` in all aspects, solving 26/51 instances and finding plans that are on average about 4 times longer, in 13 times the amount of time. In the `Metric-FF` model, we have added the same flow constraint as an additional goal conjunct $\mathit{total\_retrieved} = \mathit{total\_poured}$ but this does not help the search: `Metric-FF` is not able to solve any of the instances, giving empirical support to the idea that even if we place additional constraints on the goal, these cannot be adequately exploited by the propositional h_{FF} heuristic, *precisely because* it does not take into account the constraints induced on state variables appearing in more than one goal atom. Finally, in the PUSH domain `LAMA` outperforms in all aspects the two GBFS planners in the random variation, but shows a much poorer performance on the other variation, where the `alldiff` constraint proves its heuristic usefulness.

Overall, results show that in our test domains the use of the first-order h^{apx}_{FF} heuristic consistently results in significantly shorter plans (between approximately 1.5 to 9.5 times shorter, depending on the domain) compared to its propositional counterpart. The heuristic assessments are also more accurate in all domains, resulting in a 2.5- to 14-fold decrease in the number of expanded nodes, depending on the domain. In some cases, the increased heuristic accuracy demands significantly larger computation time, although in terms of final coverage this overhead tends to be compensated,

as the more accurate heuristic scales up better; in other cases the increased accuracy itself already allows smaller average runtimes.

3.9 Discussion

The goal in the PUSH domains can be described in FSTRIPS as placing each stone in a goal cell while ensuring that all stones are in different cells. In propositional planning, the goal is encoded differently, and the `alldiff` constraint is implicit, hence redundant. Indeed, the *flow constraint* of the FSTRIPS GARDENING domain is also redundant. The fact that the performance of `FS` is improved by adding redundant constraints reminds of CSP and SAT solvers, whose performance can also be improved by explicating implicit constraints. This distinguishes `FS` from the existing heuristic search planners that we are aware of, that *either make no room for any type of explicit constraints or cannot use them in the computation of the heuristic.* This capability is thus not a "bug" that makes the comparisons unfair but a "feature". Indeed, recent propositional planners illustrate the benefits of *recovering* multivalued (Helmert, 2009) and flow constraints (Bonet and van den Briel, 2014) that the modeler was forced to hide. Here, we advocate a different approach: to make room for these and other types of constraints at the language level, and to account for such constraints in the computation of the heuristics. We have shown how allowing function symbols at the modeling level allows us to exploit constraints that otherwise become hidden in the propositional landscape. The use of function symbols has been previously discussed in other areas of artificial intelligence such as constraint programming (Frisch et al., 2007; Marriott et al., 2008) or answer set programming (Lin and Wang, 2008; Alviano et al., 2010), but has received less attention in recent planning research, in spite of languages such as ADL supporting them (Pednault, 1989).

To summarize the results in this chapter, we have considered the intertwined problems of modeling and computation in classical planning, and found that expressiveness and efficiency are not necessarily in conflict. Indeed, computationally it may pay to use richer languages when state variables appear more than once in action preconditions and goals, as long as the resulting constraints are accounted for in the computation of the heuristic. In our formulation, heuristics are obtained from a logical analysis of the shortcomings of standard delete-free heuristics, where we have seen that the fact that the assumption of *decomposability* is valid for simple languages where formulas are conjunctions of atoms involving one state variable each, does not necessarily mean that such languages are more convenient for modeling and problem-solving than richer languages that do not enforce this restriction. On the contrary, if the goal of the problem contains different atoms involving the same state variables, a better alternative will likely be to acknowledge the constraints imposed by the atoms on these state variables *within the language itself*, so that these constraints can be used at will to derive more informed heuristics.

This has resulted in an extension of the standard relaxed planning graph into what we have called the *first-order relaxed planning graph* (FOL-RPG), where sets of values x^k that are possible for a state variable x in propositional layer x_k of the relaxed planning graph are thought of as encoding an exponential set of possible logical interpretations. Since these heuristics are more informed but intractable, we show how they can be effectively approximated with local consistency techniques. Our `FS`

planner, further described in Chapter 6, implements these ideas, along with a number of extensions to the expressiveness of the Functional STRIPS language, namely:

1. Global constraints can be freely used as (fixed-denotation) symbols of the first order language $\mathcal{L}(P)$ of any FSTRIPS problem P, which is convenient to easily model certain recurrent concepts and benefit from the increased performance of the associated constraint propagators.
2. More in general, the denotation of any fixed symbol can be defined by an external procedure. An ad-hoc constraint propagator can be defined for the symbol that makes the computation of the FOL-RPG more efficient; otherwise, the denotation of the symbol is compiled into a generic *extensional* constraint. This is convenient to model certain aspects that otherwise would be very difficult, if not impossible, to model declaratively.
3. *State constraints*, i.e. formulas that need to hold in all states of a plan, can be used at will to ease the task of modeling and to enhance the accuracy of heuristics at the same time.

The use of relaxed-planning-based ideas in settings richer than the propositional has been previously explored (Hoffmann, 2003; Gregory et al., 2012; Katz et al., 2013; Aldinger et al., 2015; Scala et al., 2016a), and actually constitutes a promising avenue of research for integrating specialized solvers into heuristic search planners, in the same manner that we have integrated a CSP solver.

Rintanen (2006) also generalizes the formulation of delete-free heuristics such as $h_{\max}$ h_{add} and h_{FF} to propositional STRIPS featuring arbitrary formulas, and identifies the NP-hardness of the satisfiability tests as one of the potential computational bottlenecks of this generalization, although his discussion is purely theoretical, and does not contemplate the use of function symbols as we do. Rintanen further explicitly identifies the *decomposition assumption* with which propositional formulations deal with this source of hardness, but argues that in general the structure of formulas is simple enough so that this theoretical hardness is not a problem in practice. This is true for the propositional benchmarks that are standard in the field, but is in general not true for formulas featuring functional symbols. The propositional (function-free) equivalent of even a simple formula such as $x < y$ will typically be complex enough, for non-trivial domains, to make an approach such as the one we present here worthwhile, as the results e.g. for the COUNTERS domain show.

Chapter 4

Planning with Existential Quantification

4.1 Motivation

Modeling and computation are essential components for using planners and many other automated tools. On the one hand, we want planners to scale up gracefully to large problems, even if the task is computationally worst-case intractable; on the other, we want users to be able to encode problems naturally and directly without having to understand how planners work. Indeed, the need to "hack" encodings to make planners work is one of the factors that hinder the use of planners by non-experts. Since modeling and computation are heavily coupled, the problem cannot be entirely solved, yet this doesn't mean that progress on this front is not possible and necessary. In the previous chapter, we analyzed how the simple addition of function symbols to the language we use to describe planning problems can have significant consequences both for modeling and computation. In this chapter, we aim at doing a similar thing, but this time discussing the use of existential quantification.

4.1.1 Effective Support for Existential Quantification

Existentially quantified variables provide indeed a simple and convenient modeling feature in problems where the goal is not a fully-specified single state, but leaves room for choices. In a Scan-like problem, for instance, the goal may be to place a set of packages in different target locations, but without explicitly saying in which target location to place each package. In the blocks-world, to name but another example, the goal may be to reach not a single, particular configuration of blocks, but a state in which a blue block is on a red block, irrespective of the actual block identities. There are no doubt other mechanisms to model such open choices in propositional languages such as STRIPS, but existential variables offer a way to do so very compactly, and in addition have the advantage of making these choices *explicit*, which is useful for reasoning about them.

As a matter of fact, existentially quantified variables in preconditions and goals are part of the standard PDDL language (McDermott et al., 1998), but few planners offer support for them, and those that do *compile them away* at a cost that is exponential in the number of this type of variables, which is definitely not a good option from a computational standpoint. We will call this approach the *compilation-based*

approach. The usual alternative in the PDDL world is to avoid existential variables altogether. In the last three International Planning Competitions (2008, 2011, 2014), for instance, no benchmark makes use of existential quantification. As we will show, however, this is not good either, as it leads to clumsy encodings that obscure the problem structure and results in less informed heuristics than the compilation-based approach, not to mention the fact that coming up with such alternative encodings is not always trivial. We illustrate these various points below.

To illustrate, consider the problem of placing a blue block on top of a red block, and this red block on top of a blue block.[1] This can be easily encoded in PDDL with a goal formula

$$\exists z_1, z_2, z_3.\, on(z_1, z_2) \wedge on(z_2, z_3) \wedge blue(z_1) \wedge red(z_2) \wedge blue(z_3).$$

Modern classical planners like `FF`, `FD`, and `LAMA` (Hoffmann and Nebel, 2001a; Helmert, 2006a; Richter and Westphal, 2010) accept such existentially quantified formulas, but they compile the quantification away at preprocessing, producing in this case a goal in disjunctive normal form (DNF) with terms

$$on(a, b) \wedge on(b, c) \wedge blue(a) \wedge red(b) \wedge blue(c)$$

for all possible combinations of blocks a, b and c. This DNF goal is mapped in turn into an atomic dummy goal that can be achieved with extra actions, whose preconditions are those terms (Gazen and Knoblock, 1997). The immediate problem with this compilation is that it does not scale up: the number of terms in the compilation is exponential in the number of variables, which is problematic for more than a few quantified variables. In our example, this will happen as soon as the desired color pattern in the goal involves more than a few blocks.

In general, it is not trivial to find equivalent and compact propositional encodings that avoid existential variables, but in some cases it is feasible. The goal of placing n packages p_i into n different target locations l_k, for instance, can be expressed with existential quantification and equality as

$$\exists z_1, \ldots, z_n \left(\bigwedge_{j \neq i} z_i \neq z_j \wedge \bigwedge_i target(z_i) \wedge at(p_i, z_i) \right).$$

The quantification and the compilation can be avoided here by using a *clear* predicate over target locations that becomes a precondition of the action of dropping a package in such locations. The resulting encoding is *equivalent* to the one obtained by compiling the previous formula in the sense that both encodings result in the same set of plans (dummy actions aside), and it has the advantage that its size does not grow exponentially with n. This compact encoding, however, is not only less clean and direct than the one with quantification, but it is also certainly *not equivalent* from a *computational* point of view. To see this, consider the behavior of the standard delete-relaxation heuristics on a situation where none of the packages is at a target location. If existential variables are compiled away as outlined above, the heuristic will estimate the cost of taking each package to a *different* target location; in the alternative encoding using the *clear* predicate with no variables, in

[1] Similar variations of the classical blocks-world have often been considered in the AI literature (Subrahmanian and Zaniolo, 1995; Hölldobler et al., 2006).

contrast, the heuristic will estimate something different: namely, the cost of taking each package to its *closest* target location. These estimates will be different when several packages have the same closest target location. This can be easily seen for $n = 2$, where, assuming the two different target locations are named l_1 and l_2, the compilation results in a DNF goal

$$(at(p_1, l_1) \wedge at(p_2, l_2)) \vee (at(p_1, l_2) \wedge at(p_2, l_1)),$$

without terms $at(p_1, l_1) \wedge at(p_2, l_1)$ or $at(p_1, l_2) \wedge at(p_2, l_2)$ where both packages end up at the same location. In contrast, the encoding with the *clear* predicate does not account for this difference, as both locations l_1 and l_2 will remain *clear* in the delete relaxation as long as they are *clear* in the state from which the heuristic is computed.

4.1.2 Constraint Satisfaction in Planning

The above discussion illustrates that while compiling away the variables in the goal has an exponential cost, *there are also hidden computational costs in the avoidance of existential variables* to prevent the exponential blow-up. This point can be made more general and compelling once we notice that existential variables allow us to *encode naturally and directly constraint satisfaction problems as planning problems.*

It is well known that CSPs and SAT problems can be encoded as planning problems. In the standard encoding, actions set the value of variables, preconditions are used to check that new values do not violate a constraint and the goal requires all variables to have a value. This "planning approach" is not expected to do well on such problems in general. Indeed, common heuristics will be useless, and the space full of dead ends (assuming that values cannot be unset, something that makes sense, as no value needs to be set more than once in a solution). In principle, the assumption that planners should not do well in SAT and CSP problems is reasonable, but this "separation of concerns" between planning and constraint-based reasoning is artificial: there are many problems that involve both planning and constraint reasoning — think e.g. on a typical graph coloring with an added agent that needs to travel around the graph, picking up and dropping a limited number of colored pencils in order to actually paint the graph vertices; or a n-queens problem where the pieces must be moved physically one square at a time until reaching a goal configuration. Several other more realistic problems sharing this double Planning-CSP affiliation have also been recently discussed, e.g. involving network-flow numerical constraints, integration of task and motion planning or scheduling of mining operations (Ivankovic et al., 2014; Mansouri and Pecora, 2014; Ferrer-Mestres et al., 2015; Burt et al., 2015).

The addition and effective support of existential variables to a language such as STRIPS provides a direct bridge between CSPs and planning. Consider for example a CSP with variables $x_1, \ldots, x_n$, domains D_i, and binary constraints $p_{i,j}$ between pairs of variables x_i and x_j expressed in extensional form such that an entry $p_{i,j}(a, b)$, for $a \in D_i$ and $b \in D_j$ means that the pair $\langle a, b \rangle$ satisfies the constraint $p_{i,j}$. The mapping of this CSP into a a STRIPS problem $P = \langle A, O, I, G \rangle$ making use of existential variables is rather straightforward:

1. P has no actions or fluent symbols, i.e. O is empty, and S contains binary atoms $p_{i,j}$ over types given by the CSP domains D_i and D_j.
2. The initial state I contains atoms $p_{i,j}(a, b)$ for each value pair $\langle a, b \rangle \in D_i \times D_j$ allowed by each constraint $p_{i,j}$.

3. The goal is simply the conjunction

$$\exists z_1, \ldots, z_n . \bigwedge_{i,j} p_{i,j}(z_i, z_j) \ .$$

where each z_i is an existentially quantified variable with type D_i.

Indeed, as we will see below, the extension of the FS planner to support existential variables handles such problems naturally, as a CSP solver, as it interfaces with the Gecode CSP solver (Gecode Team, 2006) to solve the CSPs that may arise during the computation of the heuristic and in goal and action applicability checking operations. The advantage of FS is that it is a planner, and can smoothly accommodate variations of the problem featuring actions and fluents as well, as in the agent-based graph coloring problem mentioned above, to which we return in the experimental section.

The compilation-based approach outlined above would convert such a quantified goal into a grounded DNF goal whose terms are all possible joint valuations of the CSP. The very existence of the field of constraint satisfaction bears witness to the fact that this is not an effective way of solving the CSP. The delete-relaxation heuristic for the compilation, however, is *exact*: it will yield a heuristic value of 0 for the initial state iff the CSP is actually solvable. Alternative STRIPS encodings of the CSP that do not involve variables and are compact are certainly possible, yet such compact, polynomial encodings cannot yield a (polynomial) delete-relaxation heuristic equivalent to the heuristic obtained from the compilation unless P = NP.

4.2 Overview of Results

In other words, while dealing with existential variables by compiling them away is exponential, avoiding them altogether results in poorer heuristic estimates in general. In this chapter, we show that there is actually a third option: to *explicitly handle existentially quantified variables during the computation of the delete-relaxation heuristic*. As we will show, this option avoids the exponential blow-up in the compilation, and yet it delivers a heuristic that is *equivalent to the one that would be delivered by the delete-relaxation in the grounded, compiled problem*. The proposed method is based on a modification of the relaxed planning graph (see Section 2.4.4) that *takes into account the first-order structure of the atoms*. This modification is actually an extension of the FOL-RPG and the underlying $\Gamma(\phi, P_k)$ model that we introduced in the previous chapter. This extension, however, is independent of the support for function symbols, and for most of the chapter we assume we are simply dealing with the extension of propositional STRIPS with existentially quantified variables, which we name E-STRIPS. As in the previous chapter, the computation of the heuristic is worst-case intractable, but as we will show, it can be informed and cost-effective in practice.

Interestingly, the proper handling of existential variables in STRIPS opens up a way for compiling pure FSTRIPS into E-STRIPS as in (Haslum, 2008), but with no effect on the structure and informativeness of the resulting heuristic. Another interesting corollary of this support for existential variables is that it allows *lifted planning*, i.e. planning directly with the action schemas, both in the search and in the computation of the heuristics, as action parameters are nothing else that a special kind of existential variables, as we will see.

In the rest of the chapter, we formalize the extension of propositional STRIPS with existentially quantified variables, which we name E-STRIPS (Section 4.3), and provide a minor reformulation of the FOL-RPG that accounts for these variables (Section 4.4). We discuss the link between STRIPS with functions and with existential variables, and possible compilations from functions to existentials, and the other way around, in Section 4.5. In Section 4.6 we show how the above ideas can be used to perform planning without having to ground the action schemas. We then evaluate the extension of the FS planner to support existential quantification by comparing it with state-of-the-art STRIPS planners run on existentially-quantified encodings and on alternative, compact propositional encodings that avoid the compilation, and also with an extension of the FSTRIPS planner FS, and we preliminary illustrate the performance of the planner when doing lifted planning (Section 4.7). We conclude the chapter with a discussion of our contribution and of related work. All of the work presented in this chapter, except for the section on lifted planning, which is unpublished work, has been previously published in (Francès and Geffner, 2016b).

4.3 STRIPS with Existential Quantification: E-STRIPS

Although originally conceived as a first-order language (Fikes and Nilsson, 1971), we already saw that STRIPS is most often used as a propositional language. We here extend the standard definition of propositional STRIPS given in Chapter 2 to include existentially quantified variables, and discuss how heuristics based on the relaxed planning graph can be adapted to this extension, which we call E-STRIPS.

4.3.1 E-STRIPS Language

An E-STRIPS problem is a tuple $P = \langle A, O, I, G\rangle$, with the same syntax and semantics given by the state model $\mathcal{S}(P)$ that was defined in Section 2.3.2 for STRIPS, except for one key difference: in addition to ground atoms, preconditions and goals may feature *e-formulas*. An e-formula is a first-order formula of the form $\exists x_1, \ldots, x_n.\, \phi$, where ϕ is a conjunction of atoms $p(t_1, \ldots, t_n)$ with each term t_i being either a constant symbol or a variable symbol x_i from the quantification prefix. The e-formula $\exists x_1, \ldots, x_n.\, \phi$ is true in a state s if the variable symbols x_i can be replaced by objects $o_1, \ldots, o_n$ in the STRIPS universe so that the resulting grounded formula ϕ' is true in s. This definition extends naturally for STRIPS problem with types — in such a case, variables have to be replaced by objects from the universe of the appropriate type. When $\bar{x}$ is a tuple of variables, we write the e-formula $\exists \bar{x}.\, \phi$ as $\exists \bar{x}.\, \phi[\bar{x}]$ so that the ground conjunction of atoms ϕ' that results from the substitution $\bar{x} \mapsto \bar{o}$ in ϕ can be written as $\phi[\bar{x} \mapsto \bar{o}]$ or simply $\phi[\bar{o}]$. For example, for the formula $\exists x_1, x_2, x_3.\ p(x_1, x_2) \wedge p(x_2, x_3)$, if ϕ is $p(x_1, x_2) \wedge p(x_2, x_3)$, $\phi[x_1, x_2, x_3 \mapsto c, d, c]$ denotes the variable-free formula $p(c, d) \wedge p(d, c)$.

Determining if an e-formula holds in a state s is NP-complete, meaning that plan verification in E-STRIPS is NP-complete as well, and that any heuristic h able to distinguish between goal and non-goal states (i.e., such that $h(s) = 0$ iff s is a goal state) will be intractable. This is however a worst-case result and additionally does not imply that such heuristics cannot be cost-effective, as we argue below.

4.3.2 E-STRIPS Heuristics

Adapting the relaxed planning graph heuristic from STRIPS to E-STRIPS is simple, even though the resulting heuristic is no longer polynomial. We just need to define how to deal with e-formulas in preconditions and goals in the construction of the RPG. While a ground atom p is deemed to be *satisfiable* in a propositional layer P_i when p belongs to P_i, we take an e-formula $\exists x.\phi[x]$ to be *satisfiable* in P_i *when there is a substitution* $x \mapsto o$ *such that all ground atoms in* $\phi[o]$ *belong to* P_i (recall that $\phi[x]$ is assumed to be a conjunction of ground atoms). If, for instance, ϕ is $p(x_1, x_2) \wedge p(x_2, x_3)$, then the formula $\exists x_1, x_2, x_3.\ \phi$ is satisfiable in a layer P_i that contains the atoms $p(c, d)$, $p(c, e)$, and $p(d, c)$, through the substitution $[x_1, x_2, x_3 \mapsto c, d, c]$. On the other hand, the same formula would not be satisfiable in that layer if only the first two atoms were contained in P_i.

Extending the RPG construction to take into account e-formulas is simple to state but potentially complex to compute. Indeed, the satisfiability test for e-formulas in a state or in a propositional layer P_i amounts to solving a CSP with variables that correspond to the variable symbols in the prefix, domains given by the sets of STRIPS objects (respecting type consistency), and constraints given by the atoms $p(t_1, \ldots, t_n)$ in ϕ, such that the predicate symbols represent the tables that contain the tuples $o_1, \ldots, o_n$ for atoms $p(o_1, \ldots, o_n)$ in layer P_i only. A substitution $x \mapsto o$ satisfies ϕ if o is a solution to this CSP.

For plan extraction, the supporters of an e-formula $\exists x.\phi[x]$ that is satisfied in a layer P_i with substitution $x \mapsto o$ are identified with the supporters of the atoms in $\phi[o]$. Thus the plan extraction procedure that works backward from the goal does not see e-formulas at all, just ground atoms, as in the standard RPG procedure.

We call the relaxed plans and the heuristic that result from this procedure for the E-STRIPS language $\pi^e_{\mathrm{FF}}(s)$ and $h^e_{\mathrm{FF}}(s)$ respectively. If instead of counting the actions in the relaxed plan we count the number of propositional layers, we get a variant of the $h_{\max}$ heuristic, $h^e_{\max}$, that can be shown to be admissible for E-STRIPS. The heuristic $h^e_{\mathrm{FF}}(s)$ is inadmissible, in the same way that h_{FF} is inadmissible for STRIPS.

The heuristics h^e are equivalent to the corresponding heuristics h for the STRIPS compilation and can be regarded as a *lazy* version of them. The key difference is that the compilation requires *time and space* exponential in the number of existential variables, while the heuristics for E-STRIPS require only exponential *time in the worst case*. The terms of the ground DNF formulas required for compiling away e-formulas $\exists x.\,\phi[x]$ in preconditions and goals correspond indeed to the conjunctions of ground atoms $\phi[c]$ for each of the possible substitutions $x \mapsto c$. The computation of the heuristics however does not apply these possible substitutions all at once, and moreover, it does not explore the space of all possible substitutions exhaustively either. Instead, it uses constraint propagation to search for a substitution that makes the e-formula $\exists x.\,\phi[x]$ satisfiable, as will become clear in the next section.

4.4 Supporting Existential Quantification in Functional STRIPS

If dealing with existential quantification in STRIPS is directly related to solving constraint satisfaction problems, the question naturally arises of whether the first-order RPG for Functional STRIPS that we discussed in the previous chapter, and

whose construction we mapped into a CSP as well, can be extended to support existential quantification in FSTRIPS. The answer is positive; in this section, we outline the details of such extension.

The definition of the Functional STRIPS language given in Chapter 2 does indeed already account for existential quantification. Syntactically, note that existential variables can appear in preconditions, effect conditions, and goal formulas; the scope of these quantifiers, however, does not extend to functional effect heads, since otherwise effects like $f(x) := c$ or $f(c) := x$, where x is some existentially quantified variable, could be non-deterministic, depending on the value chosen for x. Our usage is roughly analogous to the usage of the `:vars` construct in PDDL (McDermott et al., 1998) for denoting existential variables (indeed, symbol variables occurring in heads are assumed to be universally quantified). The usual way to model this in PDDL is to "push" this existential variables as extra parameters of the action schema.

The algorithm outlined in Fig. 3.4 for the construction of the first-order RPG works the same when existential quantification is allowed in the formulas, as long as the satisfying interpretations (models) in lines 1 and line 2 are understood to be extended with the actual variable substitution necessary to satisfy the respective formulas, as customary in standard first-order logic semantics (Enderton, 2001). The only thing that hence warrants discussion is how to deal with existential quantification in the constraint satisfaction problems $\Gamma(\phi, P_k)$ into which the construction of the FOL-RPG is mapped. We assume that the formula has no universal quantifiers; universal quantifiers can be eliminated by expanding them over the finite universe of their associated type. For the sake of simplicity, we further assume that all formulas are in prenex normal form, i.e. have the form $\exists \bar{x}.\phi[\bar{x}]$, where $\phi[\bar{x}]$ is a quantifier-free formula with only free variables in $\bar{x}$. Arbitrary first-order formulas can be converted to prenex normal form by variable renaming and application of simple rewriting rules.

Definition 4.1 ($\Gamma(\phi, P_k)$ with existential quantification)**.** *Let P be a FSTRIPS problem P over language $\mathcal{L}(P)$ and P_k a layer of a first-order relaxed planning graph for the problem, i.e. one domain x^k for each state variable $x \in \mathcal{V}(P)$ of the problem. The constraint satisfaction problem $\Gamma(\phi, P_k) = \langle X, D, C \rangle$ for a formula ϕ in prenex normal form* without universal quantifiers*,*

$$\phi \equiv \exists\, x_1/\tau_1, \ldots, x_n/\tau_n\, \phi[x_1, \ldots, x_n],$$

where $x_1, \ldots, x_n$ are the only free variables appearing in ϕ, contains all variables, corresponding domains and constraints specified in Definition 3.10, plus the following additional CSP variables, defined inductively over the structure of every term t appearing in ϕ:

Terms *If t is an existentially-quantified variable x with type τ, $\Gamma(\phi, P_k)$ contains a CSP variable v_x with domain given by the universe $\mathcal{U}_\tau$.*

If t is a term of the form $f(t_1, \ldots, t_m)$, and some t_i is an existentially-quantified variable, then no extra adjustment is necessary with respect to Definition 3.10. $\Gamma(\phi, P_k)$ will contain, depending on whether f if a fixed *or* fluent *symbol, an* extensional *or an* element *constraint placed upon the CSP variables $v_{t_1}, \ldots, v_{t_n}$, and v_t, which capture, respectively, the denotations of $t_1, \ldots, t_m$ and of the term t. Details are provided in Definition 3.10.*

```
type counter, value

objects c_1, ..., c_n: counter
        i_1, ..., i_m: value

predicates
  val(c: counter, v: value)
  successor(v_0: value, v_1: value)
  lt(v_0: value, v_1: value)

action increment(c: counter, v_0: value, v_1: value)
  prec val(c, v_0) ∧ successor(v_0, v_1)
  effs val(c, v_1)
       ¬val(c, v_0)

init val(c_1. 0), val(c_2, 0), ..., val(c_n, 0)

goal ∃v_1, ..., v_n. val(c_1, v_1) ∧ ... ∧ val(c_n, v_n) ∧ lt(v_1, v_2) ∧ ... ∧ lt(v_{n-1}, v_n)
```

Figure 4.1: Fragment of an E-STRIPS encoding of the COUNTERS domain (simplified syntax). Compare with the original FSTRIPS formulation in Fig. 3.3.

As in Section 3.5.2, any solution σ to the CSP $\Gamma_0(\phi, P_k)$ can be mapped into a set $\mathcal{I}(\sigma)$ of first-order interpretations consistent with the solution, which is guaranteed, by construction, to contain exactly those interpretations in $\mathcal{I}_k$ that satisfy ϕ.

4.5 Relation of E-STRIPS and Functional STRIPS

There is a close relation between E-STRIPS and Functional STRIPS. Haslum (2008) has shown how pure FSTRIPS (with no built-in functions) can be compiled into E-STRIPS by adding existential variables and replacing functions of arity m by relations of arity $m + 1$ using the principles of Skolemization in reverse. For example, a grounded atom $f(g(c)) = d$ is replaced by the e-formula

$$\exists x.\ f'(x, d) \wedge g'(c, x),$$

where the relations $f'(x_1, x)$ and $g'(c_1, x)$ represent $f(x_1) = x$ and $g(c_1) = x$ respectively. When functional terms appear in action schemas, as noted by Haslum, the existential variables that result from the above transformation can be *pushed* into the action signature as additional action schema parameters. Fig. 4.1 shows the E-STRIPS encoding that results from the application of this transformation to the FSTRIPS encoding of the COUNTERS problem that was discussed in the previous chapter, shown in Fig. 3.3. In the *increment* action, existential variables v_0 and v_1 are created representing terms $val(c)$ and $val(c) + 1$, respectively, and then they are moved to the signature of the action as extra action parameters, which by definition are (implicitly) quantified existentially. Predicates *val* and *successor* of the E-STRIPS encoding represent functions *val* and "+1" of the FSTRIPS encoding, whereas predicate *lt* represents the built-in FSTRIPS predicate "<".

This is important not only for modeling but also for computation. As we have shown in the previous chapter, the heuristics h^{FO}_{FF} and $h^{FO}_{\max}$, as well as their approximate versions h^{apx}_{FF} and $h^{apx}_{\max}$, are able to capture constraints in FSTRIPS representations of problems such as COUNTERS that are otherwise missed in their propositional versions. Interestingly, the proposed h^e heuristics for E-STRIPS are also able to leverage such constraints to gain accuracy, as they take into account the first-order structure of the formulas. As it turns out, the goal formula in the E-STRIPS encoding above, namely:

$$\exists v_1, \ldots, v_n. \bigwedge_{i=1..n} [val(i, v_i)] \wedge [\bigwedge_{i=1..n-1} lt(v_i, v_{i+1})]$$

will be satisfied in a propositional layer P_k only by the substitution that replaces the variable v_i by the value i, because the heuristic accounts for the fact that the v_i variables need to be replaced consistently in all goal subformulas. This is very interesting, as it means that the positive results obtained in Chapter 3 can be reproduced on the E-STRIPS function-less encodings of the problem obtained through Patrik Haslum's translation, *provided the new heuristics h^e are used*. Running other planners on such translations would not deliver the same results. It must be said, however, that Haslum's translation captures the core of FSTRIPS only, leaving out the ability to use the extensions outlined in Chapter 3 such as built-in functions, global constraints, externally defined symbols, etc., which on certain problems can make a large difference. On the other hand, the heuristic for E-STRIPS is simpler to describe and implement than the heuristic for FSTRIPS.

Compiling a problem represented in E-STRIPS into an equivalent problem making use of function symbols but without existential quantification is also straight-forward, although less elegant. We do not provide the full details here, but the intuition is that every distinct existential variable x of type τ in an action precondition or in the goal formula can be modeled by adding

1. An extra type τ' with universe $\mathcal{U}_{\tau'} = \mathcal{U}_\tau \cup \{u\}$, where u is a new element representing the *undefined* value.

2. An extra constant v_x with type τ' (which will be a state variable of the problem).

3. An extra action schema with signature $\texttt{set}_x\texttt{(o: } \tau\texttt{)}$, with no precondition and with effect $v_x := o$.

The initial state is then extended by assignments $v_x := u$, and each occurrence of x can then be replaced by the (fluent) constant v_x, adding, where necessary, a condition specifying $v_x \neq u$. This compilation can be seen as representing the bindings of existential variables to propositions (Kautz et al., 1996). Plans for this compilation can be readily converted into plans for the original E-STRIPS problem by removing the $\texttt{set}_x$ actions. The two encodings will not however have the same computational properties.

4.6 Lifted Planning: Planning without Grounding Action Schemas

The interpretation of existential variables so far presented provides also a principled way for dealing with action schemas without grounding them either in the search or in the computation of the heuristic. This is critical in some applications where action schemas take many arguments and where exhaustive grounding is not feasible (Koller and Hoffmann, 2010; Masoumi et al., 2013; Areces et al., 2014). Exhaustive grounding of action schemas is not too different indeed from grounding goals with existential quantifiers: if the action preconditions are constrained, there might be several valuations for the action arguments, but few of them that actually satisfy the preconditions and thus make the grounded action applicable in a given state. In general, a large number of ground actions also imposes a heavy load during the construction of the relaxed planning graph.

As we have already pointed out, Functional STRIPS often allows a reduction of the number of ground actions thanks to the use of (possibly nested) function symbols, as in the case of the N-PUZZLE, that can be modeled with 4 ground actions. In contrast, the standard STRIPS encoding requires more than n^2 ground actions. Still, the language and the computation model, which relies heavily on consistency tests performed by an underlying CSP solver, suggest alternative ways for dealing with action schemas without having to ground them.

We need to distinguish between the use of action schemas during the computation of the relaxed plan and during the search. In the the construction of the planning graph, the action schema parameters can be treated *exactly as existential variables* with types given by the type of the schema parameter. In other words, an action schema a with 6 argument variables whose types include 10 constants each, would result in 10^6 ground instances. In this lifted method for computing relaxed plans, on the other hand hand, the six argument variables are added to the corresponding CSPs, and dealt with efficiently by using constraint satisfaction techniques.

Doing lifted planning during the search, on the other hand, is different than lifting in the computation of the RPG, as one cannot "choose" the action schema but has to generate, evaluate, and select particular ground instances. In our context of forward search, the main implication of this is that we need to be able to compute, given any state s, and for each action schema a the *set of groundings of the schema that make the schema applicable* in that state. For this, one needs to build a single CSP per action schema. The different solutions of this CSP can then be directly associated to the different ground instances that are applicable in any given state, and to the state variable updates that they lead to. Another way of understanding this approach is that the grounding of action schemas is done "on the fly", i.e. by solving, each time that a state s is expanded, a constraint satisfaction problem which is built based on the action parameters, action precondition and on the state s. Each solution to this CSP gives a possible variable substitution σ that is equivalent to an action grounding. Interestingly, this CSP *is exactly* the constraint satisfaction problem $\Gamma_0(pre(a), P_0)$, where P_0 is the initial RPG layer for state s or, alternatively, the layer that allows for each state variable $x \in \mathcal{V}(P)$ of the problem, the singleton set of values $\{x^s\}$.

In a sense, the only difference between grounding all actions statically before the search and doing the search with action schemas as discussed here is that the first approach is nothing but an exhaustive generate-and-test method applied at pre-

processing time, whereas the second can apply forms of constraint reasoning in order to speed up the search, but is applied during search time. Computing all the solutions of a CSP is more complex than testing satisfiability, and the variables in this action schema CSP are typically few. If that is the case, it may be better to ground the actions previous to the search, whereas if the number of action parameters is large, the CSP approach is likely to scale up better. On their discussion of operator splitting techniques in the context of reductions of planning to propositional satisfiability, Kautz et al. (1996) discuss ideas similar to the ones discussed here. Younes and Simmons (2002) also discuss performance differences between planning with lifted and grounded actions, but in the context of partial order planning.

4.7 Empirical Evaluation

4.7.1 Setup

Most of the results reported in this section were first published in (Francès and Geffner, 2016b). The aim of the section is to analyze empirically the different ways in which problem representations using existential variables can be dealt with. To this end, we consider four families of problems, each of which we encode in three different formalisms:

1. E-STRIPS, i.e. propositional STRIPS with existential quantification *but no function symbols*. As noted above, this is included in the PDDL standard.

2. Propositional STRIPS, where existential variables have been removed by some manual reformulation of the problem that preserves equivalence.

3. Functional STRIPS with existential quantification.

The methods proposed in this chapter have been implemented in the `FS` planner. A full account of the current capabilities of the planner can be found in Chapter 6. The planner interfaces with the `Gecode` constraint solver (Gecode Team, 2006) in order to compute the heuristics based on the FOL-RPG that we have described above, thus offering full support for both problems represented in E-STRIPS and in FSTRIPS with existential quantification, as the former is a subset of the latter. The discussion of the results of the planner when applied to E-STRIPS encodings and to FSTRIPS encodings are kept separately, to emphasize that the proposed heuristics are useful even when only STRIPS with existential quantification is considered. As we will see, however, the additional compactness of FSTRIPS pays off also computationally when paired with existential quantification. In both cases, the planner uses a plain greedy best-first search coupled with the h^e_{FF} heuristic.

The performance of the `FS` planner running on all the above encodings is compared against that of the `Fast-Downward` planner (Helmert, 2006a, `FD`) running on those encodings which it accepts: E-STRIPS and propositional STRIPS. `FD` is configured with the same search strategy as `FS`: a greedy best-first search with the h_{FF} heuristic, a single queue, no EHC, and non-delayed evaluation. All planners run a maximum of 30 minutes on a cluster with AMD Opteron 6300@2.4Ghz nodes, and are allowed a maximum of 8GB of memory. The source code of the `FS` planner plus all problem encodings are available on `www.gfrances.github.io`.

4.7.2 Domains

The four families of domains that we test are BW-PATTERN, VERTEX-COLORING, GROUPING and COUNTERS, the last two of which were already discussed in Chapter 3. We briefly describe them here:

Blocksworld Pattern BW-PATTERN, sketched in the chapter introduction, is a BLOCKSWORLD variation where each block has a color, and we want to reach *any* block configuration displaying a given color pattern, such as "a red block on a blue block on a green block", modeled in E-STRIPS with the goal formula

$$\exists\, b_1, b_2, b_3.\ col(b_1, red) \wedge col(b_2, blue) \wedge col(b_3, green) \wedge on(b_1, b_2) \wedge on(b_2, b_3).$$

In FSTRIPS, locations and colors of the blocks are modeled with a function symbol instead of a predicate, but the goal formula remains otherwise equal. The FSTRIPS encoding results however in more compact action schemas, and avoids the need of modeling block moves with a sequence of pick and place operations. For BW-PATTERN we test no propositional encoding, as it is not trivial to reformulate the problem without existential variables.

Vertex Coloring VERTEX-COLORING illustrates how any CSP can be easily embedded in an (action-less, fluent-less) E-STRIPS planning problem. A classical k-VERTEX-COLORING problem with n nodes and set of edges E maps into an E-STRIPS problem with k objects of type τ_c (i.e. of "color" type) and typed goal e-formula

$$\exists\, z_1/\tau_c, \ldots, z_n/\tau_c \bigwedge_{(i,j)\in E} z_i \neq z_j.$$

The initial (and only) state of this planning problem will be detected as a goal if and only if the graph is k-colorable. The above formula is also the one we use as the FSTRIPS encoding, as in this case there is no gain to be obtained through the use of function symbols. We have also devised a propositional STRIPS reformulation with a single action paint(v, c) that paints vertex v with color c if it was previously unpainted. The action cannot be applied if this would create an invalid vertex painting. This is encoded with a precondition subformula involving universal quantification:

$$\forall v'/\tau_v(E(v, v') \rightarrow unpainted(v', c))$$

where $E(v, v')$ is the graph edge relation, τ_v is the type that represents "graph vertex" objects, and $unpainted(v', c)$ denotes that vertex v' is *not* painted with color c. Nodes are initially all unpainted, and a plan consists in a sequence of node paints (whose ordering is actually irrelevant).

We also test a variation of the domain that we name AGENT-COLORING, where an agent moves around the graph and has to manually perform the painting of the vertices by picking up some cans of paint that are randomly distributed, with the additional constraint that only one such can can be carried at a time. This variation requires few changes, but provides the chance of giving a more planning-like nature to the problem, in the sense that the order of actions becomes relevant.

We use two sets of instances: first, random instances ($100 \leq n \leq 500$, $10 \leq k \leq 30$) where the graph results from adding edges at random to a uniform spanning tree until a certain graph density is reached; second, standard instances from a public compilation from the literature,[2] of which we have pruned those instances reported as not solvable in less than 1 hour by state-of-the-art coloring methods *or* having a chromatic number $\chi > 10$, From the remaining 34 coloring instances, we generate planning problems with a number of colors $k \in \{\chi, \chi + 1\}$, for a total of 68 non-random instances.

Grouping Blocks In GROUPING, already discussed in Chapter 3 (see e.g. Fig. 3.6), colored blocks are randomly placed on a grid and need to be moved around with the goal that two blocks end up on the same cell iff they have the same color. The original FSTRIPS formulation features a goal formula with an atom $loc(b_i) \bowtie loc(b_j)$ for each pair of blocks b_i, b_j, where $\bowtie$ is $=$ ($\neq$) if the two blocks have the same (different) color. This translates directly to E-STRIPS, so that the goal of a problem with red blocks a and b and a blue block c is

$$\exists\, l/\tau_L, l'/\tau_L.\; l \neq l' \wedge at(a, l) \wedge at(b, l) \wedge at(c, l'),$$

where τ_L is the type corresponding to grid locations.

The propositional STRIPS reformulation that we have devised uses additional actions to "label" in advance the final grid location for the blocks of each color, and once the target location of all colors is fixed, it moves blocks to their corresponding color locations. The E-STRIPS encoding that we just sketched, however, is not only much simpler and cleaner, but it preserves the problem structure in a manner which allows the h^e_{FF} heuristic to be more informed than h_{FF}, as we discuss below.

Counters Finally, COUNTERS, also described in the previous chapter (see e.g. Figs. 3.2 and 3.3) is the problem were we have n integer variables and apply actions to increase or decrease their value by one until the inequalities $x_1 < \ldots < x_n$ hold. We use two variations, labeled COUNT_0 and $\text{COUNT}_{\text{RND}}$, where variables are initially set to 0 and to random values, respectively. The E-STRIPS encoding is the one shown in Fig. 4.1, which is the result of applying Patrik Haslum's compilation technique to the original FSTRIPS encoding, plus replacing the "$<$" FSTRIPS built-in predicate for a lt predicate with extensional denotation.

The propositional encoding, in turn, requires the use of conditional effects and a linear number of extra atoms to model when two consecutive variables x_i, x_{i+1} satisfy the $x_i < x_{i+1}$ goal requirement. An alternative encoding could feature existential variables quantified in pairs, i.e. with a goal formula

$$\bigwedge_{i=1..n-1} \left(\exists z, z'.\, val(c_i, z) \wedge val(c_{i+1}, z') \wedge lt(z, z')\right).$$

As long as the problem actions maintain the functional character of the val relation, this formulation is equivalent to the one presented in Section 4.5. We expected `Fast-Downward` to have less difficulties here during preprocessing time, even if at the cost of a certain loss of heuristic accuracy, but it turns out that it is the other way round, the planner barely going beyond preprocessing on any instance, so we omit results for this encoding from the discussion.

[2] https://sites.google.com/site/graphcoloring/, accessed on 31 July 2017.

Domain	N	Coverage			Plan length			Node expansions			Time (s.)		
		E-FF	P-FF	E-FS	E-FF	P-FF	E-FS	E-FF	P-FF	E-FS	E-FF	P-FF	E-FS
COUNT$_0$	12	3	**7**	**7**	-	70.7	70.7	-	542K	70.7	-	52.6	84.9
COUNT$_{\text{RND}}$	36	9	**25**	21	21.4	21.6	22.1	22.4	36.6	22.1	0.9	0.0	0.7
GROUPING	48	28	34	**47**	25.8	41.6	26.1	26.8	203K	26.2	2.1	179.5	54.6
BW-PATTERN	30	11	-	**20**	8.4	-	50.9	306.3	-	16.19K	0.6	-	60.9
V-COLORING	80	1	43	**78**	-	84.3	0	-	60.7K	0	-	87.7	0
A-COLORING	88	6	**30**	27	24.5	26.9	29.2	41.2	80.8	65.0	0.0	0.0	0.3

Table 4.1: **E-STRIPS planning**. Comparison of results between different planners on E-STRIPS and propositional STRIPS encodings. "E-FF" is the FD planner with FF-like configuration on E-STRIPS encodings; "P-FF" is the same planner on a (manual) propositional STRIPS reformulation (except for BW-PATTERN, for which we have no such encoding); "E-FS" is our FS planner on the E-STRIPS version; N is number of instances; length, node expansion and time figures are averages over instances solved by all those planners that solve at least 5 instances. Best-of-class coverage numbers are shown in bold.

Domain	N	Coverage			Plan length			Node expansions			Time (s.)		
		P-FF	E-FS	EF-FS	P-FF	E-FS	EF-FS	P-FF	E-FS	EF-FS	P-FF	E-FS	EF-FS
COUNT$_0$	12	7	7	**11**	70.7	70.7	70.7	542K	70.7	70.7	52.6	84.9	5.6
COUNT$_{\text{RND}}$	36	25	21	**30**	93.0	86.6	86.2	6.9K	87.2	86.2	15.8	117.9	11.2
GROUPING	48	34	47	**48**	42.4	23.8	24.3	150K	23.8	24.3	146.8	25.9	5.8
BW-PATTERN	30	-	20	**29**	-	45.1	5.7	-	9.28K	6.2	-	38.8	0.2
V-COLORING	80	43	**78**	**78**	84.3	0	0	60.7K	0	0	87.7	0	0
A-COLORING	88	30	27	**38**	71.9	94.9	77.9	353.3	1.82K	609.6	0.1	30.9	4.1

Table 4.2: **E-STRIPS *vs.* FSTRIPS planning**. Comparison of results between different planners on E-STRIPS, propositional STRIPS and FSTRIPS encodings. "P-FF" is FD with FF-like configuration on a (manual) propositional STRIPS reformulations (except for BW-PATTERN, for which we have no such encoding); "E-FS" is our FS planner on E-STRIPS encodings; "EF-FS" is our FS planner run on FSTRIPS encodings that feature existential variables, when necessary. P-FF and E-FS columns are from Table 4.1. N is number of instances; length, node expansion and time figures are averages over instances solved by the three planners. Best-of-class coverage numbers are shown in bold.

Instance	Plan length				Node expansions				Time (s.)			
	FF	FS_0	FS_1	FS_2	FF	FS_0	FS_1	FS_2	FF	FS_0	FS_1	FS_2
PUSH-RND $(7, 4)$	44	39	39	**32**	244	64	64	**42**	**0.03**	0.67	0.68	0.72
PUSH-RND $(10, 6)$	108	**100**	**100**	**100**	1319	**526**	**526**	672	**0.35**	48.97	26.23	36.93
PUSH-RND $(13, 8)$	288	-	247	**239**	37554	-	**1417**	2717	**25.34**	-	231.23	447.44
AVERAGE	87.71	59.96	59.96	**54.11**	3131.39	315.39	315.39	**304.04**	**0.98**	68.76	21.57	23.05

Table 4.3: **Planning without Action Grounding.** Results on the random PUSH domain, including three selected instances plus the average over the 36 instances in the benchmark set. PUSH-RND (n, k) denotes grid size n^2 and k stones. Results for planners FF and FS, the last running in three different modes: standard (FS_0), RPG-lifted (FS_1) and Fully-lifted (FS_2) (see text for a full description). A dash indicates time or memory out. Best-of-class numbers shown in bold typeface. FS_2 solves 34 instances, FS_0, 30, and FF and FS_1, 32 instances each.

4.7.3 Results on E-STRIPS Encodings

Table 4.1 shows the results of the FS planner run on E-STRIPS encodings (labeled "E-FS"), as well as of Fast-Downward on both E-STRIPS encodings (labeled "E-FF") and on manual propositional reformulations (labeled "P-FF"). We next highlight the main conclusions that can be drawn from these results.

First, *the exponential time and space required to compile away E-STRIPS existential variables has a huge impact on domain coverage.* In general, FD tends to either time or memory out during preprocessing. As an example, the planner times/memories out on GROUPING instances with more than 3 colors, and on COUNTERS instances with more than 9 variables. The only domain with acceptable coverage is GROUPING, but this is just because the benchmark set includes many instances with only 2 existential variables.

Second, *the heuristic that can be derived from the propositional reformulations is less informed than the one computed from the E-STRIPS encoding.* In GROUPING or in COUNT_0, for instance, the average number of node expansions is orders of magnitude higher on the propositional reformulation, because the delete-relaxation does not account for certain constraints that are captured on the E-STRIPS version, such as (in GROUPING) the constraint that all blocks of the same color must be on the same position *at the same time.* In spite of this, however, FD has much better performance on these propositional reformulations than on the E-STRIPS encodings, because it avoids the exponential time and space required to compile away the existential variables and to process the result of the compilation.

Third, *the first-order approach of FS to handle existential variables avoids the exponential time and space penalty of the compilation approach without making the resulting heuristics less informed*, making it a more effective strategy, and resulting in consistently higher coverages. When compared to FD on E-STRIPS, the average number of nodes expanded by FS is similar in all domains but in BW-PATTERN (more on this below). In VERTEX-COLORING, instances are large, and FD on E-STRIPS solves only one instance, likely because the planner takes too long determining which of the exponential number of ground actions into which the goal e-formula is compiled are applicable In contrast, FS needs only perform one single call to the underlying CSP solver, therefore scaling up to large instances in less than 1 second.

Fourth, *when compared to the performance of FD on the propositional reformulations, FS produces better coverage on two cases and worse coverage on two other cases*, excluding the BW-PATTERN domain that is not simple to encode propositionally. The average number of expanded nodes is in general much lower for FS, up to one or two orders of magnitude in some cases, but at the same time node expansion rate is also around between one and two orders of magnitude faster in FD. Upon inspection, FS suffers a significant overhead caused by the lack of multivalued variables in E-STRIPS, which results in CSPs with too many Boolean variables that could well be replaced by fewer multivalued variables. This shortcoming does not appear in FSTRIPS, which in addition benefits from built-in functions and constraints. We next discuss the results on FSTRIPS.

4.7.4 Results on FSTRIPS Encodings

Indeed, one of the advantages of FSTRIPS is that it in general results in more compact encodings. This is true not only at the level of the declarative model, but also at

the level e.g. of the CSP encodings that are used to compute the FOL-RPG on which the heuristics are based. Table 4.2 compares FD on the propositional STRIPS reformulations ("P-FF", which is the best option for Fast-Downward according to Table 4.1, leaving aside the reformulation effort that it requires) with FS on E-STRIPS encodings ("E-FS") and with FS on equivalent FSTRIPS encodings that exploit both functions and existential variables ("EF-FS"). In some cases (COUNTERS, GROUPING), these are pure FSTRIPS encodings that do not need existential quantification, but in the remaining domains (BW-PATTERN, VERTEX-COLORING, AGENT-COLORING) existential quantification is indeed necessary.

When compared to FS on E-STRIPS encodings, FS on the FSTRIPS encodings shows higher node expansion rates and even a more informed heuristic in BW-PATTERN. This is because

1. Functional STRIPS encodings usually result in fewer grounded actions.
2. The CSPs used during the heuristic computation have significantly fewer variables (e.g. quadratically so in the case of GROUPING).
3. These CSPs benefit from custom constraint propagators for FSTRIPS built-in symbols, such as the "$<$" symbol in COUNTERS.
4. Certain tie-breaking mechanisms during the plan extraction phase of the FOL-RPG are easily implemented for FSTRIPS, but have not yet been implemented for existential variables in the current version of the FS planner, which upon inspection seems to be harming heuristic accuracy particularly in BW-PATTERN.

Overall, the FS planner is more performant when working on a combination of FSTRIPS with existential quantification than when working on E-STRIPS reformulations alone, offering a coverage which consistently dominates the other two planners, and plan length, average number of expanded nodes and overall running time figures which are consistently better. In particular, the extension of FSTRIPS with existential quantification yields better results than the manual propositional STRIPS reformulations *in all domains*.

4.7.5 Results on Lifted Planning

In order to test whether the above ideas on planning without grounding the action schemas do actually work well in a problem with a potentially large number of ground actions, we have run a preliminary experiment in the random variation of the PUSH domain described in Section 3.8, which is actually a simplification of the well-known SOKOBAN domain where grid obstacles have been lifted. We compare the FF planner with three different execution modes of FS, labeled $\texttt{FS}_0$, $\texttt{FS}_1$ and $\texttt{FS}_2$:

- $\texttt{FS}_0$ is the standard version of the planner that fully grounds the problem actions at preprocessing time.
- $\texttt{FS}_1$ uses only action schemas for the construction of the first-order relaxed planning graph, but grounds the actions at preprocessing and performs the forward search with ground actions.
- $\texttt{FS}_2$ performs fully-lifted planning, i.e. uses action schemas both in the construction of the first-order relaxed planning graph and during the forward search.

Table 4.3 shows the results of this comparison. Out of 36 instances, coverage (not shown in the table) is similar for the four planners: $\texttt{FS}_0$ solves 30 problems, $\texttt{FS}_2$, 34,

and the remaining two solve 32 instances each. Thus, the three FS variants remain competitive with the FF planner, which runs significantly faster due to its cheaper heuristic, but this cheaper heuristic results in a significantly higher number of node expansions as well. More relevant are the relative differences between the three FS variants. It turns out that the heuristic is roughly equally informed in the three cases, but when it comes to runtime, $\texttt{FS}_1$ and $\texttt{FS}_2$ are significantly faster then the standard $\texttt{FS}_0$, which is the cause of the higher coverage we just mentioned. Interestingly, what makes a difference is not so much avoiding action grounding altogether during the search, but using action schemas in the construction of the relaxed planning graph. This is due to the fact that heuristic computation times heavily dominate overall search time, being roughly responsible for 90% of it. Altogether, the characteristics of the PUSH domain (a high number of ground actions with highly constrained preconditions – at most 8 actions can be applicable in SOKOBAN at any time, 4 moves and 4 pushes) appear to be a good use case where both the reduction of action arguments allowed by nested fluents *and* the ability of FS to compute heuristics and perform search directly with action schemas pay off computationally. This however remains to be tested over a larger set of benchmarks as future work.

4.8 Discussion

This chapter presents a novel view of existential quantification in STRIPS that relates planning to constraint satisfaction and relational databases (Gottlob et al., 1999). A goal that involves existential variables is indeed like a query; what makes it particular in the context of planning is that we are not simply evaluating the goal "query" in a *static* database, but can apply actions that modify the database in order to make the query satisfiable, and hence the goal true.

We have also argued that the use of existential quantification for leaving some choices open in goal and preconditions allows for simpler and more concise models, while giving planners the chance to reason about those choices. Instead of compiling existential variables away with an exponential technique or avoiding them altogether, we have shown how, as in the previous chapter, delete-relaxation heuristics can be extended to deal with existential variables by using constraint satisfaction techniques, resulting in heuristics that are more informed than those that can be obtained from alternative propositional encodings.

This view of existential variables in STRIPS has resulted in the extension of he existing FSTRIPS FS planner to offer full support for both E-STRIPS and FSTRIPS with existential quantification. This is achieved by extending the first-order relaxed planning graph defined in the previous chapter to support existential quantification. This extension requires only a few, simple modifications to the constraint satisfaction problems into which the construction of such relaxed planning graph is mapped. This simplicity, however, comes hardly as a surprise, as the formulation of the language, the semantics, and the computational model are not propositional but first-order.

We have developed a small number of benchmark domains and encoded them in different representational formalisms (propositional STRIPS, E-STRIPS, FSTRIPS with existential variables) in order to empirically evaluate the proposed methods. The fact that the best computational results have been achieved for the more expressive language illustrates the importance of making problem structure explicit so that it can be exploited computationally (Rintanen, 2015; Ivankovic et al., 2014). The work

we have presented here is also related to (Porco et al., 2011, 2013), where a general compilation mechanism is presented to compile formulas for the existential fragment of second-order logic, which includes the existential fragment of first-order logic, into STRIPS.

The use of existential quantification, in particular, is also related to the identification of symmetries in planning problems (Fox and Long, 1999), and might open up novel ways of dealing with them effectively. Riddle et al. (2015a,b, 2016) provide a mechanism to automatically reformulate PDDL problems to reduce symmetries, by way of what the authors call *bagged representations*.[3] Here we focus on providing a higher-level language in which such reformulation mechanisms are less necessary because *the language provides the adequate means to represent the same problem in a more convenient manner*. In the case of GRIPPER, for instance, a language that can deal with integer variables such as FSTRIPS can help reduce symmetries.

[3] In GRIPPER, for instance, the identities of the different balls in the problem are completely irrelevant, and the problem can be reformulated to keep track of the number of balls in every room instead of the exact room where every individual ball lies.

Chapter 5

Planning with No Language

5.1 Motivation

Research in planning during the last decades has studied a wide variety of planning models, from classical (deterministic, fully-observable) to temporal models, Markov decision processes, partially-observable MDPs, etc. What seems to be common in the area, however, is that between the mathematical model and the actual problem that falls within the space defined by that model, there is a fundamental third entity: a declarative modeling language, e.g. STRIPS or PDDL, that bridges the gap between the abstractness of the model and the concreteness of the particular problem.

Planning languages serve several important roles, among which:

1. They describe *in a compact manner* problems whose size is usually exponential with respect to the encoding.
2. They are *general* enough so that different types of problems and domains can be fed into planners.
3. They reveal *problem structure* that can be exploited computationally, as we have already seen in previous chapters.

Indeed, the planners that address the simplest (classical) planning models and scale up best all exploit in some way or another the problem structure encoded in action preconditions, effects, and goals. This includes from the first means-end and partial-order planners (Newell and Simon, 1963; Tate, 1977; Nilsson, 1980; Penberthy and Weld, 1992) to the latest SAT, OBDD, and heuristic search planners (Kautz and Selman, 1996; Edelkamp and Kissmann, 2009; Richter and Westphal, 2010; Rintanen, 2012). Key techniques in modern planning such as heuristic estimators (McDermott, 1999; Bonet and Geffner, 2001b), helpful actions (Hoffmann and Nebel, 2001a), and landmarks (Hoffmann et al., 2004), are all computed from the delete-free relaxation of the *problem representation*, but *cannot be derived from the underlying planning model alone*.

The focus on standard, declarative planning languages has resulted in a dramatic progress in scalability in spite of the intractability of the classical planning problem (Bylander, 1994), but has a downside too: some problems that clearly fit e.g. the classical planning model are not easy at all to represent in declarative classical planning languages such as PDDL. The canonical example are problems that require some geometrical reasoning, but this shortcoming has been exposed more

acutely with the recent development of *simulators* that pose interesting challenges as potential stepping stones towards truly general artificial intelligence (McCarthy, 1987; Pennachin and Goertzel, 2007). Simulators are software architectures that encode classical or nearly-classical planning models (known initial state, deterministic actions, and, often, reachability goals) *in an implicit manner*, i.e., through some programmatic interface; however, as it is not at all clear how to extract a declarative model from these simulators, existing classical planners cannot be directly used for planning in them. Recent simulators include the Atari Learning Environment (Bellemare et al., 2013), the games of the General Video Game competition (Perez-Liebana et al., 2016), Minecraft (Johnson et al., 2016) and the Universe platform (OpenAI, 2016).

In Chapters 3 and 4, we have argued in favor of planners that support more expressive, first-order languages, not only because they allow more natural and elegant models, but also because effective reasoning strategies can be employed when problems make use of that additional expressiveness. In this chapter, we explore an orthogonal way of dealing with the same powerful modeling features, e.g. function symbols, existential quantification, externally-denoted symbols, conditional effects or derived predicates. Instead of extending existing heuristic constructs to support those features, which is costly, we directly develop a planning method that actually *does not care about the type of features used to represent the model*, or, for that matter, about the declarative representation of the problem at all. We call this classical planning method *planning with simulators* or *simulation-based planning*, as virtually the only thing that is required is a *simulator* that *implements* the classical planning model as described in Section 2.2. In particular, the proposed method makes no assumption nor places any requirement on the form of the transition function $f : S \times O \mapsto S$. Indeed, our method treats the transition function as a *black box*, and concepts such as *the problem language*, the *precondition of an action*, or *the effects of an action*, which hitherto were central to the planning enterprise, become largely irrelevant. As we will detail below, only a factorization of the state into state variables and, for increased performance, a count of the number of goal conditions achieved in any state, need to be provided, in addition to the constructs specified by the classical planning model.

This of course does not need to be taken as implying that modeling languages are not important. On the contrary, it is as an additional argument in favor of more expressive modeling languages that allow the modeler to easily and succinctly represent the problem she has in mind without having to worry about the potential performance impact of the language constructs that she uses. Domain-independent planners able to plan with simulators rather than planning languages can thus be a key addition to the planning toolbox while, broadening the scope of planners. Such planners would be insensitive to the syntax of action descriptions, since they would not see such descriptions at all (Vallati et al., 2015b). Indeed, the result of applying an action to a state can be obtained from declarative descriptions or procedures, whatever is more convenient, reducing the challenge of modeling.

The key question we address in this chapter is: is it possible to achieve the goal of *planning with simulations* while *retaining the scalability* of planners that are based on and make use of the actual declarative representations of actions? In other words, can a domain-independent planner that has access to the *structure of states* and to *some notion of progress towards the goal* only approach the performance of planners

that have also access to the *structure of actions*? To address this question, we first formalize the theoretical definition of simulators Section 5.2. In Section 5.3, we give a brief overview on the most recent state-of-the-art *width-based methods* upon which our work is based, which were already introduced in Section 2.4.6, and then go on to design a domain-independent family of classical planning algorithms that ignore the representation of actions (Section 5.4). We perform an experimental evaluation of our methods over standard planning benchmarks in Section 5.5, and find out that they can match state-of-the-art performance in spite of not performing any reasoning based on action structure. We explore the exciting possibilities that open up for modeling and for expressing domain-dependent knowledge when planning languages are replaced by simulation languages in Section 5.6, and conclude the chapter with a general discussion of the contribution in Section 5.7. Most of the work presented in this chapter has been previously published in (Francès et al., 2017).

5.2 Factored State Models and Simulators

A *classical planner* is a program whose input is a *compact representation* of a classical planning model and whose output is a plan. Compact representations are usually expressed in a declarative planning language such as STRIPS, PDDL or ADL, but these purely declarative planning languages are not the only way to represent classical planning models in compact form. State models $\Pi = \langle S, s_0, S_G, O, f \rangle$ can also be represented in general by using state variables and by encoding the partial transition function $f(a, s)$ (and, when relevant, the cost function $c(a, s)$) as a *black box* procedure. This is indeed what many simulators do. For instance, in the ALE and GVG-AI simulators, the transition function f is encoded by a procedure, all actions are always considered to be applicable, and the cost function $c(a, s)$ encodes rewards (negative costs). One problem with this approach is that it is not general. There is indeed no way for simulating the blocks-world in the ALE. On the other hand, the GVG-AI simulator has some generality, as it comes with a language for a class of games that is partially declarative and partially procedural (Perez-Liebana et al., 2016). Such a representation of actions has been seldom used in domain-independent planning, and in what follows we address the question of whether effective, general planning methods can be developed for it. To this end, we first need to introduce the concept of *factored state model*:

Definition 5.1 ((Classical Planning) Factored State Model)**.** *A (classical planning)* factored state model *is a tuple* $\mathcal{F} = \langle V, D, s_0, G, O, f \rangle$ *that consists of*

- *A finite set of* state variables $V = \{x_1, \ldots, x_n\}$.
- *A finite domain* $D(x)$ *for each state variable* $x \in V$.
- *An initial assignment* s_0 *of values to state variables which is compatible with their domains.*
- *A set* $G = \{g_1, \ldots, g_m\}$ *of* goal conditions *expressed as Boolean functions* $g_i : D(x_1) \times \ldots \times D(x_n) \mapsto \{\top, \bot\}$.
- *A set of actions or operators* O.
- *A (partial) transition function* $f : S \times O \mapsto S$.

As usual, we will denote by $A(s) \subseteq O$ the set of operators applicable in state s:

$$A(s) = \{o \in O \mid \exists s' \; f(s, o) = s'\}$$

The tuple $\mathcal{F}$ provides a compact representation of a corresponding state model $\mathcal{S}(\mathcal{F}) = \langle S, s_0, S_G, O, f \rangle$ where S is the set of variable assignments compatible with their domains, and S_G is the set of assignments satisfying all goal conditions in G. The goal conditions in G do not have to be atoms of the form $X = x$, but can be *arbitrary* procedures mapping states into Booleans. A factored state model is not a logic-based language such as FSTRIPS, but for the sake of consistency, we will still often refer to "atoms" in the context of factored state models, by which we will understand any fact of the form $x = c$, where $x \in V$ is a variable of the problem and $c \in D(x)$ one of its allowed values.

Any STRIPS planning problem $P = \langle A, O, I, G \rangle$ can be converted into a factored state model $\mathcal{F} = \langle V, D, s_0, G', O', f \rangle$ where

- The set of state variables is $V = F$,
- the domains of all state variables are Boolean,
- s_0 is the assignment that assigns the value $\top$ to those state variables whose corresponding atom appears in I, and $\bot$ otherwise,
- $O' = O$, and
- the transition function f is derived in linear time from the information in action preconditions and effects in the standard manner (see e.g. Section 2.3.2).

This simple and polynomial translation is not bidirectional. In what follows, a *simulator* will be a factored state model $\mathcal{F} = \langle V, D, s_0, G, O, f \rangle$ where the function f and all goal condition functions $g \in G$ are given by black box procedures. The focus on black box methods that plan with simulators does not mean that these functions cannot be described declaratively or through hybrids involving procedures, but rather that simulation-based planning algorithms make no assumption about the form of such descriptions.

5.3 Width-Based Methods and BFWS(f)

The methods developed for planning in simulated environments like ALE and GVG-AI are based mainly on blind and heuristic search, Monte Carlo Tree Search (MCTS, Browne et al., 2012), width-based search, and hybrids of these (Bellemare et al., 2013; Perez-Liebana et al., 2016; Soemers et al., 2016). Blind search methods can be applied directly to factored state models that just encode weighted directed graphs in compact form. Heuristic search methods, on the other hand, require heuristics, which are not easy to derive from simulations. MCTS methods potentially combine the benefits of blind and heuristic search: they can be used off-the-shelf as the former, but scale up better as value functions akin to heuristic functions are incrementally learned and used to guide the search. Usually, however, MCTS performs this bootstrapping slowly when rewards are sparse and there is no domain-dependent knowledge in the form of informed base policies. This explains why MCTS is not used in classical planning.

Other planning methods based on the concepts of *novelty* and *width* of a problem, however, have been successfully used both in simulated environments like ALE and

GVG-AI and in classical (propositional) planning (Lipovetzky and Geffner, 2012; Lipovetzky et al., 2015; Geffner and Geffner, 2015; Shleyfman et al., 2016; Jinnai and Fukunaga, 2017). In Chapter 2 we described the basic IW(k) algorithm and the so-called *Best-First Width Search* (BFWS) methods (Lipovetzky and Geffner, 2017a,b) that work on STRIPS encodings and build on the same notion of novelty to perform an effective exploration of the state space which, when coupled with information extracted from standard classical planning heuristics, is able to match the performance of state-of-the-art planners such as `LAMA`, `Mercury` or `Jasper` (Richter and Westphal, 2010; Katz and Hoffmann, 2014; Xie et al., 2014a). These BFWS methods perform a greedy best-first search prioritizing search nodes lexicographically according to the novelty of the node, in the first place, and to some standard heuristic measure, to break ties. BFWS algorithms often do not use the vanilla definition of novelty, but instead use the notion of *novelty w_F of a state given certain functions*, which in effect partitions the whole state space into different equivalence classes determined by the value of certain *state functions* $f \in F$, $f : S \mapsto \mathbb{N}$, where S is the set of states of the problem (see Section 2.4.6 for details), and then the novelty of each newly-generated state s is computed *with respect to those previously-seen states that belong to the same equivalence class than* s, i.e. share the same f-values.

The best BFWS planner in (Lipovetzky and Geffner, 2017a), named BFWS(f_5), however, does not completely fit with the description above, as it uses *path-dependent* (instead of *state-dependent*) metrics to partition the search space. The novelty metric used by BFWS(f_5) is $w_{\{u,r\}}$ (we will use the denotation $w_{u,r}$ when there is no possible ambiguity), where

- The function u maps states s to the number of atomic goals $u(s)$ that are *unsatisfied* in s.[1]
- The function r is a subtler *path-dependent* metric that roughly tries to capture progress towards achieving *potential subgoals* not explicitly represented in the problem goal, and involves the computation of a (delete-free) relaxed plan.

More precisely, when a state s is generated that achieves more goals than its parent p (i.e. $u(s) < u(p)$), a *set of atoms* $R(s)$ is computed. For any descendant s' of s reached from s through intermediate states that do not achieve more goals than s, $r(s')$ is the number of atoms in $R(s)$ that are made true *at some point* in the way from s to s'. For example, if $R(s)$ is defined as a set of landmarks, then $r(s')$ would count the number of those landmarks in $R(s)$ achieved in the way from s to s', even if they do not hold in s' anymore. This definition of the function r is generic for any possible definition of the set of atoms $R(s)$; in BFWS(f_5) in particular, $R(s)$ is defined as the set of all atoms appearing in the preconditions and positive effects of the actions in a *relaxed plan* $\pi_{\text{FF}}(s)$ computed from s (Hoffmann and Nebel, 2001a).

The greedy best-first search in BFWS(f_5) then orders search nodes by the lexicographic evaluation function $f(n) = \langle w_{u,r}, u \rangle$, i.e., prioritizing nodes n with lower value $w_{u,r}(n)$, and breaking ties in favor of nodes with lower value $u(n)$ and, eventually, of nodes with lower accumulated cost to the node.

Example 5.2. *To illustrate, each time a search node n is generated in BFWS(f_5), its novelty $w_{u,r}$ is computed. To that end, the (state-dependent) value of $u(n)$ and*

[1] This assumes, as is usually the case in benchmarks from the International Planning Competitions, that the goal is specified as a conjunction of ground atoms.

the (path-dependent) value of $r(n)$ and need to be computed. $u(n)$ is the number of goal atoms that have yet not been achieved in the state corresponding to the node n, and can be computed in a straightforward manner. The computation of $r(n)$ depends on a set $R(s')$, which is determined as follows:

1. *If $u(n) < u(p)$, where p is the parent node of n, then we take s' to be the state that corresponds to search node n. In that case, a relaxed plan π_{FF} is computed from s' with the standard relaxed planning graph procedure, and any atom appearing on a precondition or a positive effect is collected into $R(s')$.*

2. *If, on the contrary, $u(n) \geq u(p)$, then we seek the closest ancestor n' of n such that $u(n) < u(n')$, and take $R(s')$ as the basis for computing $r(n)$, where s' is the state that corresponds to search node n'.*

Whatever is the case, $r(n)$ is computed by traversing all the nodes from s' to s, both inclusive, and counting how many atoms in $R(s')$ are true in some state. The novelty value $w_{u,r}$ is then computed by following Definition 2.16, i.e. taking into account only those previously-seen nodes n' that have the same values $u(n') = u(n)$ and $r(n') = r(n)$.

5.4 Simulation-Based Planning with BFWS(R)

The BFWS(f_5) algorithm is a state-of-the-art method for classical planning, yet it cannot be applied to simulations. This is because the computation of the sets of atoms $R(s)$ used in the definition of the path-dependent r counter relies on the delete-relaxation of the problem, which is not available from simulations. In order to have an algorithm that can plan with the black box functions $A(s)$ and $f(s, a)$ (action costs are assumed to be 1), we generalize the BFWS(f_5) procedure into a *family* of search algorithms, called BFWS(R), that differ from BFWS(f_5) only in the way that the sets of atoms $R(s)$ are defined and computed. Thus, BFWS(R) is a best-first search algorithm with a primary evaluation function given by novelty measures $w_{u,r}$, breaking ties using the u counter and accumulated costs. The BFWS(f_5) algorithm is BFWS(R) with $R(s)$ defined as the set of atoms in a relaxed plan computed from the state s, which we will denote as $R_X(s)$. Other methods for defining and computing $R(s)$ are explored below, most of which do not require declarative action descriptions. Moreover, we focus only on methods that define and compute the set R *once* from the initial state s_0 and then fix $R(s)$ to $R(s_0)$ for any other state s where the set R is required. This is because the computation of such sets, while polynomial, can be expensive.

The intuition behind the BFWS(R) search schema, where R is a fixed set of atoms precomputed from the initial state, is that R represents a set of "potential subgoals" that, along with the given set G of goal conditions, partitions the search nodes into classes associated with different "subproblems"; namely, the set of nodes that satisfy the same subset of goals from G and have reached the same set of subgoals from R. In each subproblem, the aim is to reach another goal or another potential subgoal that can eventually lead to another goal, ideally, by expanding novelty-1 nodes only (of which there is a linear number). Yet, since the number of such subproblems is exponential in the sizes of G and R, that objective is approximated by merging together subproblems with the same *number* of goals and subgoals u and r. This

Algorithm: Computation of R_G from state s_0

for $k = 1, 2$ **do**
  Run IW(k) from s_0 and set N_g to contain all nodes n in the search where $g(n)$ is true, for each goal condition $g \in G$
  if $N_g \neq \emptyset$ *for all* $g \in G$ **then**
    Let N_g^* be a set with one *arbitrary* node $n \in N_g$ for each $g \in G$
    Let Π_G^* contain all paths from s_0 to some node in N_g^*
    return a set with all atoms that are true at some point in some plan in Π_G^*
return R_A

Figure 5.1: Computation of the goal-oriented set of atoms R_G.

is indeed what the novelty measure $w_{u,r}$ used in BFWS(R) does.[2] In particular, a novelty measure of 1 for a state s means that s is the first state in the search that makes some atom p true, among the states s' generated so far that belong to the same "subproblem" as s, namely, that have the same u and r counts. The result is that the total number of subproblems is given by $|G| \times |R|$, and hence the maximum number of states that can have novelty k (given functions u and r) is $|G| \times |R| \times |F|^k$, where $|F|$ is the number of problem atoms.

The discussion and definitions below will be mostly tested in problems that are encoded in propositional STRIPS, but apply in general to any problem which can be represented as a factored state model, in which case by "atom" we understand some equality $x = c$, where $x \in V$ is a variable of the problem and $c \in D(x)$ one of its allowed values. The choices of the sets R of atoms in the general algorithm BFWS(R) that we consider are all *fixed*, in the sense that each is computed *once* from the initial state s_0 as a *polynomial form of preprocessing* from the simulation, so that for any state s where the set $R(s)$ is required, $R(s) = R(s_0)$. The different sets R that we consider are:

1. R_0 is the empty set. For this set, $r(s) = 0$ for all s.
2. R_A is the set of all atoms; i.e., $R_A = F$.
3. $R[k]$ contains all atoms are true in some state that is reachable from s_0 by running IW(k), for $k \in \{1, 2\}$.
4. R_G is a goal-oriented version of $R[k]$ that is obtained from the plans to achieve from s_0 some individual goal condition of the problem. These plans are computed by IW(1) and IW(2) alone, as detailed below.
5. R_G^* is a variant of R_G that skips IW(2) computations altogether when the number of ground actions is too large.
6. R_X is the union of the preconditions and positive effects of the actions in a *relaxed plan* computed from s_0 (Lipovetzky and Geffner, 2017a). This set *cannot* be computed from simulations and is included only as a baseline.

[2] A related discussion can be found in Lipovetzky (2014).

The exact definition of R_G is given in Fig. 5.1. The procedure IW(1) is run from s_0; if that single invocation of IW(1) finds partial plans that reach each one of the individual goal conditions of the problem,[3] then R_G is set to the collection of atoms made true at some point in the execution of such partial plans. If, on the contrary, there is some goal condition for which IW(1) does not find a plan, then IW(2) is run from s_0. If IW(2) finds partial plans for each of the goal conditions, R_G is set to the collection of atoms made true by such plans. Otherwise, i.e. if neither IW(1) nor IW(2) reach each of the goal conditions, R_G is set to R_A; i.e., the collection of all atoms.

This definition takes advantage of the fact that the *width* of many standard domains when the goals are atomic is often 1 or 2, meaning that IW(1) or IW(2) will find plans for them in low polynomial time (Lipovetzky and Geffner, 2012). The requirement that IW(1) and IW(2) reach *all* of the goals in order to consider the atoms appearing in state trajectories reaching a goal is there just to keep things simple. Indeed, one could consider the atoms in such trajectories even if, say, 10% of the goals are not reached. Also for simplicity, when each of the problem goals is reached by IW(1) or IW(2) through more than one plan, we collect in R_G the atoms made true by only the first plan found for each goal, discarding all the others, which in principle could also yield useful information.

The definition of R_G is reminiscent of the use of relaxed plans in PDDL planning. Indeed, the union of the plans found by IW(k) that reach each of the individual problem goals is a plan for a problem relaxation which is tighter than the standard delete-relaxation: whereas the relaxed plan for a goal $G_1 \wedge \ldots \wedge G_n$ in the delete-relaxation is the union of *relaxed* plans for each of the G_i goals, the relaxed plans computed by IW(k) are made up of *actual* plans for each G_i. The downside of this is that IW(k) will not deliver any individual plan if the width of the atomic goals G_i is higher than k, and that even for a value such as $k = 3$, IW(k), while polynomial, can be prohibitively expensive (Lipovetzky and Geffner, 2012).

The set R^*_G is defined exactly as the set R_G except for problems involving too many ground actions ($> 40,000$), where running IW(2) becomes too expensive. For such problems, R^*_G is set to R_G when the IW(1) run reaches all of the problem goals, but when not, the IW(2) computation is skipped and R^*_G is set directly to the collection of all atoms R_A.

Notice that the first three R options, R_0, R_A and $R[k]$, are independent of the problem goal, i.e. are blind search methods, while the last three, R_G, R^*_G, and R_X, are all goal-oriented, with R_X being the only option that assumes knowledge of action preconditions and effects. For the other R sets, BFWS(R) is a simulation-based planning method.

5.5 Empirical Results

In this section we report on the performance of the simulation-based BFWS(R) family of planning algorithms, for the different possible definitions of the set R that we discussed above. Performance figures are given in Tables 5.1 and 5.2. The first table

[3] Finding plans that reach each of the goals *individually* is different than finding plans that reach all goals *jointly*. By partial plan here we mean a path in the search state from s_0 to some state s that satisfies (at least) one goal condition.

	PDDL Planners												Simulation Planner			
	FF*				LAMA-11				BFWS(f_5)				BFWS(R^*_G)			
	C	L	T	Exp	C	L	T	Exp	C	L	T	Exp	C	L	T	Exp
Barman (20)	11	195	1K	5M	19	219	20	42K	**20**	174	21	107K	9	226	369	1M
Caved. (20)	**7**	23	71	1M	**7**	23	117	1M	**7**	24	0	8K	**7**	24	3	44K
Childs.† (20)	**7**	65	391	278K	5	69	3	2K	2	50	372	406K	5	57	176	59K
CityCar (20)	11	39	3	11K	5	36	631	1M	5	29	153	222K	**19**	30	57	27K
Floort.† (20)	**2**	38	5	169K	**2**	39	21	246K	**2**	42	5	73K	0	-	-	-
GED† (20)	0	-	-	-	**20**	135	4	3K	18	126	29	17K	**20**	133	10	8K
Hiking (20)	**20**	58	3	26K	18	54	284	36K	16	51	150	176K	7	64	246	174K
Maint.† (20)	10	116	3	16K	0	-	-	-	**16**	86	38	86	**16**	85	94	1K
Openst.† (20)	0	-	-	-	**20**	892	31	892	**20**	840	262	65K	14	780	1K	123K
Parking (20)	13	97	433	18K	**20**	105	114	2K	**20**	105	205	3K	**20**	106	397	44K
Tetris (20)	10	61	412	76K	10	60	615	59K	11	70	152	9K	**20**	61	158	5K
Thought. (20)	10	72	24	190K	16	76	2	673	17	72	3	5K	**20**	70	19	8.0K
Transp. (20)	4	325	147	15K	12	233	58	6K	**20**	240	6	39K	**20**	234	141	43K
Visitall† (20)	0	-	-	-	**20**	4K	212	21K	**20**	3K	52	4K	**20**	3K	10	3K
Elevat.† (20)	0	-	-	-	**20**	229	145	19K	**20**	247	20	35K	18	334	516	51K
Nomyst. (20)	9	29	234	1M	11	28	0	929	**14**	30	3	43K	13	30	63	76K
Pegsol (20)	**20**	31	4	97K	**20**	33	2	18K	**20**	29	109	785K	**20**	29	5	123K
Scanal. (20)	**20**	50	19	595	**20**	42	25	508	**20**	40	5	342	**20**	40	6	429
Sokob. (20)	18	231	34	487K	**19**	240	122	817K	15	195	122	2M	14	178	322	5M
All (380)	*172*	*0.94*	*0.60*	*0.59*	*264*	*0.92*	*0.51*	*0.35*	***283***	*0.87*	*0.27*	*0.35*	*282*	*0.89*	*0.48*	*0.45*

Table 5.1: **Performance of PDDL Planners vs. Best Simulation Planner**. Coverage (C), avg. plan length (L), avg. runtime in sec. (T), and avg. number of expanded nodes (Exp). Averages computed over instances solved by all planners, except in domains marked with †, where there are less than three commonly-solved instances. Times are total and include preprocessing. Best coverage in bold. Last row shows aggregated coverage and, for L, T and Exp, the avg. of *normalized values* computed for each planner over all domains, except those marked with †, where an L, T, and Exp value x is normalized by dividing it by the maximum value of that attribute over all planners (i.e., lower is better).

	BFWS(R_X)				BFWS(R_0)				BFWS(R_A)				BFWS(R[1])				BFWS(R_G)			
	C	L	T	Exp	C	L	T	Exp	C	L	T	Exp	C	L	T	Exp	C	L	T	Exp
Barman†	6	161	759	7M	0	-	-	-	8	245	75	319K	**16**	271	109	342K	9	237	391	1M
Caved.	**7**	23	6	265K	**7**	23	8	141K	**7**	26	6	79K	**7**	23	6	91K	**7**	24	3	44K
Childs.†	0	-	-	-	0	-	-	-	7	54	364	141K	**8**	56	345	196K	5	57	181	59K
CityCar	4	28	86	184K	15	26	37	35K	5	28	253	488K	5	26	176	176K	**18**	27	57	28K
Floort.†	**1**	42	578	14M	0	-	-	-	0	-	-	-	0	-	-	-	0	-	-	-
GED	**20**	148	39	269K	**20**	122	17	13K	**20**	126	5	8K	**20**	127	9	8K	**20**	133	28	8K
Hiking†	**16**	51	84	116K	2	42	12	142K	6	70	316	209K	3	44	1K	4M	7	76	525	241K
Maint.	14	83	293	582	**16**	83	122	1K	**16**	83	104	1K	**16**	83	99	1K	**16**	83	109	1K
Openst.†	**20**	839	112	63K	0	-	-	-	11	769	1K	124K	2	822	2K	140K	11	778	2K	120K
Parking	**20**	87	7	2K	**20**	102	441	51K	**20**	110	451	46K	**20**	110	480	46K	16	109	1K	46K
Tetris	17	93	145	261K	**20**	76	419	17K	**20**	78	230	10K	**20**	94	316	17K	**20**	84	307	11K
Thought.	17	84	8	172K	15	95	14	8K	**20**	87	71	69K	**20**	89	392	251K	**20**	86	187	46K
Transp.	14	248	21	63K	7	233	664	368K	17	224	51	39K	**20**	231	63	41K	7	290	613	59K
Visitall	**20**	3K	5	4K	**20**	3K	9	3K	**20**	3K	10	3K	**20**	3K	12	3K	**20**	3K	22	3K
Elevat.	18	175	19	27K	16	178	314	126K	15	282	148	46K	12	670	372	119K	**20**	261	348	30K
Nomyst.	**19**	24	0	4K	6	24	7	100K	10	26	14	124K	12	25	1	8K	13	24	3	16K
Pegsol	**20**	29	4	81K	**20**	30	3	100K	**20**	30	4	111K	**20**	30	4	111K	**20**	29	4	123K
Scanal.	**20**	40	1	358	**20**	40	6	432	**20**	40	6	430	**20**	40	5	430	**20**	40	7	429
Sokob.	**15**	186	106	3M	13	178	422	11M	12	184	183	4M	13	182	164	3M	13	191	233	4M
All	***268***	*0.89*	*0.36*	*0.56*	*217*	*0.87*	*0.62*	*0.66*	*254*	*0.91*	*0.53*	*0.58*	*254*	*0.95*	*0.56*	*0.56*	*262*	*0.92*	*0.68*	*0.44*

Table 5.2: **Performance of BFWS(R) algorithms for different R sets**. These are all simulation-based algorithms except for BFWS(R_X), that makes use of action structure and is included as a baseline. See caption of Table 5.1 for an explanation of the different entries.

compares the top-performing R, R^*_G, against two state-of-the-art PDDL planners, LAMA and BFWS(f_5), and a version FF* of the FF planner (Hoffmann and Nebel, 2001a) that is supported in the Fast Downward planner, where FF's heuristic is used to drive a best-first search with two queues, one restricted to the nodes obtained with helpful actions only (Helmert, 2006a). FF is a second-generation heuristic search planner that uses helpful actions, while LAMA is a third-generation heuristic search planner that uses a landmark heuristic as well. Together, FF and LAMA have been among the top-performing classical planners for the last 15 years. The original version of FF could not be used, as many instances include cost information. For the experiments, however, actions costs are taken to be 1 so that plan cost is equal to plan length. The second table evaluates BFWS(R) for the different choices of R, including the baseline option, $R = R_X$, that relies on PDDL encodings for computing a relaxed plan once from the initial state.

Benchmarked problems include all instances from the last planning competition (IPC 2014), along with all instances from IPC 2011 domains that did not appear in the 2014 IPC, with the exception of *Parcprinter*, *Tidybot* and *Woodworking*, which produced parsing errors. There are thus a total of 19 domains, with 20 instances each, for a total of 380 instances. All planners and configurations in both tables have been run on AMD Opteron 6378@2.4Ghz CPUs with CPU-time and memory cutoffs of 1h and 16GB respectively. Tables report the standard measures: coverage (number of instances solved), and average plan lengths, (total) runtimes, and number of node expansions. Average times are in seconds rounded to the nearest integer. All averages are over the instances solved by all planners except when there are less than three such instances. Implementation details of the BFWS planners are given in Chapter 6; LAMA and FF* are run using the latest version of Fast Downward (Helmert, 2006a) available at the time of running the experiments.

Table 5.1 shows that the best simulation-based BFWS(R) planner, BFWS(R^*_G), performs extremely well in spite of not using any information about action structure, when compared to the state-of-the-art PDDL planners. Indeed, while LAMA solves 264 of the 380 instances (69.5%), BFWS(R^*_G) solves 282 (74.2%). The PDDL planner BFWS(f_5) does only slightly better, solving 283 (74.5%). FF*, on the other hand, solves 172 instances (45.3%), which illustrates that the considered problems are challenging. The planners, however, do very differently in the different domains. For example, in the domains Barman, Hiking, and Openstacks, LAMA solves 19, 18, and 20 instances respectively, while BFWS(R^*_G) solves 9, 7, and 14. On the other hand, in domains such as CityCar, Maintenance, and Tetris, LAMA solves 5, 0, and 10 instances, while BFWS(R^*_G) solves 19, 16, and 20. Surprisingly, these numbers are better than those of BFWS(f_5) that solves 5, 16, and 11 instances, suggesting that relaxed plans are not helping that much in such domains. This seems to be confirmed by the other PDDL planner, R_X, shown in Table 5.2, that solves 4, 14, and 17 of the instances. In terms of average plan lengths and times, there is no clear pattern, although BFWS(f_5) is usually fastest, and along with LAMA, is the one that expands fewer of nodes. The averages of normalized plan lengths, times, and number of expanded nodes, displayed at the bottom of both tables, show that the simulation planner BFWS(R^*_G) is only a bit slower than LAMA, produces plans of similar quality and, somewhat surprisingly, does not expand too many more nodes on average, in spite of the strong exploratory nature of BFWS algorithms.

Table 5.2 evaluates BFWS(R) for different alternatives of R. While none of them

is better than R^*_G in terms of coverage, they all do better than FF*, with some of them approaching the performance of LAMA. Except for R_X, they are all simulated algorithms that do not exploit action descriptions. In comparison with the simulation planners, the relaxed plan computed by R_X appears to provide useful guidance in domains such as Hiking, Openstacks, and Nomystery, where simulation planners solve much less instances. In fact, in Nomystery R_X solves many more instances than LAMA and BFWS(f_5), 19 vs. 11 and 14 respectively, meaning that the computation of all the other relaxed plans does not pay off. Since R^*_G is R_G except in instances that have more than 40,000 ground actions, it is only on those instances where the performance of BFWS(R^*_G) and BFWS(R_G) differ. These instances come mainly from the Parking and Transport domains, where the latter planner solves 16 and 7 instances, while the former solves them all. The empty R set, R_0, is the weakest R, with 217 instances solved, well behind the other options and LAMA, although well ahead of FF*, which solves 172 instances.

The domains where simulated BFWS(R) algorithms struggle are of two types: problems where the atomic goals have a *high width*, and problems with *tightly constrained* plans where goals and subgoals are *not easy to serialize*. Examples of the former class of domains include Barman, Hiking, and Openstacks. Examples of the latter include Cavediving, Childsnack, and Floortile. This last class of domains, like any tight *constraint satisfaction problem* encoded as a planning problem, are hard also for heuristic PDDL planners like LAMA and BFWS(f_5), which nevertheless perform much better in the first class of problems due to the exploitation of action structure in the form of relaxed plans.

5.6 New Possibilities for Modeling and Control

We have seen how a move from purely declarative planning languages to simulation languages that freely combine declarative representations with procedures does not entail a significant performance loss, as one can plan effectively by exploiting state and goal structure while ignoring action structure. We now illustrate what can be gained from this move by considering two other aspects: modeling and control knowledge.

5.6.1 Modeling

Many classical problems that require reaching a goal by applying deterministic actions from a known initial state cannot be modeled easily using the standard planners, sometimes because of limitations of planners, that do not always provide good support to complex language constructs (Ivankovic and Haslum, 2015), sometimes because of limitations of planning languages. Challenging domains include, for instance, so-called "pseudo-adversarial" problems involving adversaries that follow a known, deterministic strategy that is nonetheless difficult to express propositionally, and problems like Pong, where physical actions have complex ramifications, like a ball bouncing against walls. Many of these limitations, however, can be overcome *in one shot* through the combination of expressive modeling languages and black box planning algorithms. A modeling language like Functional STRIPS offers the best of the declarative and procedural worlds; namely, a declarative, first-order logical language for modeling, where the denotation of fixed function and predicate symbols can be given extensionally, through sets of atoms, or intensionally, by means of

```
action move(x_1: loc)
  prec alive(p) ∧ @valid_move(loc(p), x_1)
  effs loc(p) := x_1
       ¬pellet_at(x_1)
       pellet_at(x_1) → C := C + 1
       ∀g ∈ ghost [loc(g) := @move_G(loc(g), x_1)]

goal alive(p), C = K
init alive(p), C = 0, pellet_at(l_1), ...
def alive(p: pacman) ≡ ∀g : ghost loc(g) ≠ loc(p)
```

Figure 5.2: Fragment of Pacman encoding in FSTRIPS where the goal is to eat K food pellets. Function and predicate symbols $@f$ denote functions and relations specified by external procedures. The encoding requires a single action *move*, which notably uses (a) a conditional effect to update a "counter" C of collected food pellets, and (b) a set of universally-quantified effects to move all ghosts as dictated by the (deterministic) external procedure $@move_G$. See text for further details.

procedures. This distinction makes no difference to simulated planning algorithms like BFWS(R) that get to see only the black box action transition and applicability functions.

Figure 5.2 shows a possible FSTRIPS encoding of Pacman. In Pacman, the agent has to collect a number of food pellets (dots) by moving around in a maze, while avoiding a number of deadly ghosts that chase him. Functional (predicate) symbols $@f$ denote (Boolean) functions specified by external procedures. In this case, the maze is implicitly encoded by the Boolean function *@valid_move*, and the deterministic strategy followed by the ghosts (move always to the adjacent grid position that is closest to the pacman according to the Manhattan-distance) is encoded by the $@move_G$ procedure, where ghosts are assumed to know where the pacman is moving. For simplicity, "power pellets" are omitted. We have run different versions of BFWS(R) in these and other domains that are difficult or impossible to tackle with PDDL planners. These include a single-agent version of the *Pong* videogame, where the agent controls a paddle and aims at hitting a ball with billiard-like dynamics a certain number of times; *Trapping* (Ivankovic and Haslum, 2015), where the agent has to trap an opponent (a cat) that is always looking for the nearest exit; *Helping*, in which the agent guides the cat to an exit by turning lights on that the cat will follow if sufficiently close, and *Pursuit*, a predators-and-prey game. All problem encodings can be found at https://github.com/aig-upf/2017-planning-with-simulators.

Interestingly, the GVG-AI competition problems encoded in VGDL, an elegant language for encoding grid-problems that is part declarative and part procedural (Schaul, 2013), should be easy to translate into FSTRIPS, once dynamic object creation and non-determinism are excluded. Non-deterministic and probabilistic extensions of FSTRIPS were implemented early in the GPT planner, that deals with MDPs and POMDPs (Bonet and Geffner, 2001a).

5.6.2 Control Knowledge: Features and BFWS(F)

Features offer a way to express domain-dependent knowledge in the context of width-based search algorithms like IW and BFWS. A feature ϕ is a Boolean function of the state which can be used as an extra atom in the computation of novelty. For example, in blocks-world problems, a Boolean feature $\phi_{x,y}$ can be defined for each goal atom $on(x, y)$, that is true in a state s when $clear(x)$ and $clear(y)$ are both true in s. By adding such a feature, BFWS(R_0), for example, decrements the number of nodes expanded to reach the goal by an average factor of 3. Such a reduction happens because some relevant states that are reached in the search with novelty 2, will have novelty 1 once the extra atoms are considered. This is indeed what conjunctive features do: they promote or give priority to states that achieve certain conjunctions of atoms. Yet features can be arbitrary Boolean functions of the state, not just conjunctions, and an interesting generalization can be obtained by combining *features* and explicit *priorities.*

Let a *ranked feature list* be a list $F = \langle F_1, \ldots, F_M \rangle$ where F_i is a set of Boolean features with priority i. Ranked feature lists provide an *abstraction* and *generalization* of the algorithms IW and BFWS, where the notion of novelty is decoupled from the problem atoms and the size of conjunctions. For this, let the *F-novelty* of a state s in the search, $w_F(s)$, be the minimum i such that for some feature ϕ in F_i, ϕ is true in s for the first time in the search, with the convention that $w_F(s)$ is $M + 1$ when there is no such feature.

The IW(1) algorithm can be seen as a breadth-first search where states with novelty $w_F(s) > 1$ are pruned, where $F = \langle F_1 \rangle$ and F_1 is given by the features $\phi_{\langle p \rangle}$, one for each problem atom, such that $\phi_{\langle p \rangle}(s)$ is true iff p is true in s. Similarly, IW(2) is a breadth-first search where states with novelty $w_F(s) > 2$ are pruned, where $F = \langle F_1, F_2 \rangle$, and F_2 contains a feature $\phi_{\langle p,q \rangle}$ for each pair of different atoms p and q, and $\phi_{\langle p,q \rangle}(s)$ is true iff p and q are both true in s.

More interestingly, BFWS(R) for $R = R_0$ (empty R) is a plain best-first algorithm guided by a novelty function w_F where the features are $F = \langle F_1, F_2 \rangle$, and F_1 and F_2 contain the features $\phi_{\langle p,i \rangle}$ for each atom p, and the features $\phi_{\langle p,q,i \rangle}$ for each pair p, q, where $1 \leq i \leq |G|$. The feature $\phi_{\langle p,i \rangle}$ is true in a state s when p is true in s and $u(s) = i$, while $\phi_{\langle p,q,i \rangle}$ is true in s when both p and q are true in s, and $u(s) = i$. Finally, the *general* BFWS(R) algorithm can be *approximated* when the features in F_1 and F_2 are defined as $\phi_{\langle p,i,j \rangle}$ and $\phi_{\langle p,q,i,j \rangle}$ where in both cases the new j component, $1 \leq j \leq |R|$, tests whether the number of atoms in R that are true in s, $r'(s)$, is equal to j. This is an approximation because the r' counter is state-dependent, while the actual counter r used in BFWS(R) is path-dependent (counts number of atoms from R reached in the way to s).

The algorithms IW(F) and BFWS(F) refer to the IW and BFWS algorithms where novelties are computed according to the ranked feature list F. The use of these lists not only provides a uniform way for understanding and programming width-search based algorithms, but also opens up the possibility of encoding control knowledge by playing with features and priorities, which could potentially be inferred from training data using machine learning algorithms.

5.7 Discussion

In this chapter we have presented a new class of black box planning algorithms. From the General Problem Solver to the latest, state-of-the-art SAT-based and heuristic search planners, planning algorithms have been predominantly based on deriving useful information from the declarative representation of the problem actions (precondition, effects). In contrast, the algorithms we have presented here, based on the recent notions of state novelty and problem width, have knowledge only about to the structure of states (i.e. its decomposition into state variables), the initial state, and the decomposition of the problem goal into different goal conditions, but *completely ignore the information about actions*. In spite of that, we have shown that they approach the performance of the best classical planners over the existing PDDL benchmarks. This suggests that the computational role of declarative planning languages such as PDDL may be overrated.

Declarative languages, however, remain important from the point of view of modeling. Indeed, as observed by Haslum (2011) in the context of `TLPlan`, the fact that our planner only requires a black-box implementation of the transition function to generate successor states implies that the language can include powerful and expressive modeling features such as externally-denoted symbols, derived predicates, etc. Planning languages such as Functional STRIPS provide the best of both worlds, allowing compact first-order constructs and, when necessary, procedurally-defined symbols. While this expressive power is not compatible with mainstream planning methods, it can be exploited by the methods proposed here. Additionally, the use of features and of ranked feature lists opens up a novel way in which domain-dependent control knowledge that helps speed up the search (Junghanns and Schaeffer, 1999; Bacchus and Kabanza, 2000) can be manually specified or learned. Effective black box planning methods can thus produce a radical change in the scope and use of planners, and in the ways in which planning problems are modeled.

Chapter 6

The FS Planner

From a practical perspective, one of the most important outcomes of the work carried out during the elaboration of this thesis is the FS planner, where every one of the ideas that have been discussed so far have been implemented and tested. The planner has been in constant evolution since the work on this thesis began. While most of the design and development of the planner has been carried out by myself, credit is due to Miquel Ramírez for contributing to very significant parts of the planner, in particular to the integration with the Gecode constraint solver, to important improvements in the grounding module of the planner (see below), and to the efficient implementation of novelty-related algorithms, on which matter the help of Nir Lipovetzky has also been invaluable.

The planner is open-sourced under a GNU General Public License, version 3 (GPL-3.0), and available for download at https://github.com/aig-upf/fs. Different parts of the planner make use or are built on top of the following third-party software components:

- The parsing of FSTRIPS encodings is sometimes performed with (a modified version of) the Python PDDL parser component of the Fast-Downward planner (Helmert, 2006a), available at www.fast-downward.org.
- Some of the search and width-based algorithms are taken from or strongly inspired in the Lightweight Automated Planning Toolkit (Ramirez et al., 2015), available at http://lapkt.org
- The computation of the first-order RPG on which the heuristics described in chapters Chapters 3 and 4 depend, as well as the lifted planning search mode, is mapped into a number of CSPs solved by the Gecode constraint programming framework (Gecode Team, 2006), available at www.gecode.org.
- The grounding of certain propositional encodings is (optionally) performed by mapping it into a logic program solved by Clingo, the Answer Set solver from the Potsdam Answer Set Solving Collection (Gebser et al., 2012), available at https://potassco.org.

In this short chapter, we discuss the architecture of the planner, review some of the language extensions it supports and then go to discuss some key optimizations geared towards achieving better performance. Some future development lines of the planner are also discussed in Chapter 7.

6.1 Design Overview

The FS planner can be best thought of as divided into a front-end, implemented in Python, and a back-end, implemented in C++. Broadly speaking, the Python front-end is in charge of the following tasks:

1. To parse the problem representation. The planner supports the full specification of the FSTRIPS language plus some extensions described in Section 6.2. Standard PDDL benchmarks usually employ a subset of the previous, and are thus also supported.
2. In certain cases, to perform an efficient, logic programming-based grounding of the problem, as detailed in Section 6.3.1.
3. To generate an internal representation of the problem used by the core C++ planner routines, optionally including grounded operators (see previous point).
4. If the problem has externally-defined symbols, the front-end compiles the definition of these symbols (which must be provided as a set of C++ classes conforming to a certain interface) and links it with the core planner library to produce a standalone, instance-specific binary. Otherwise, a generic binary is used.
5. To invoke the corresponding binary, generic or otherwise, to bootstrap the actual planning process.

The C++ back-end, which contains most of the code and functionality, broadly follows the design of the Lightweight Automated Planning Toolkit (Ramirez et al., 2015), and is thus divided into distinct modules for *language support*, *planning models*, *search algorithms* tackling those models, *heuristic evaluators*, and *state representations*. In more detail, the most relevant modules of the back-end include:

1. A *FSTRIPS language module* that contains abstract syntax trees for the full language, as described in chapters Chapters 2 to 4, plus adequate devices to represent and reason with action schemas, ground actions, state constraints, etc. Given that FSTRIPS is a superset of (basic) STRIPS, this module is able to deal with the latter as well.
2. A built-in *naive* action *grounder*, described in Section 6.3.1.
3. An implementation of the *successor generator* strategy described in (Helmert, 2006a) to iterate through applicable actions in the set $A(s)$ in a performant manner, see Section 6.3.4.
4. A set of *generic search algorithms* making heavy use of C++ templates for the sake of performance (Burns et al., 2012), which include breadth-first search, greedy best-first search, IW(k), and BFWS(R).
5. A *constraint satisfaction* module that interfaces with the C++ libraries of the Gecode constraint solver in order to set up and solve the constraint satisfaction problems $\Gamma(\phi, P_k)$ that arise during the computation of the first-order RPG, as described in Chapter 3. Optimization details are given in Section 6.3.3.
6. A set of *heuristic evaluators* which include evaluators for the heuristics based on the first-order relaxed planning graph, as described in Chapters 3 and 4.

7. A set of *novelty feature evaluators* that map any state into a set of novelty features, including the trivial state-variable evaluator ϕ^0 (Section 2.4.6) as well as more sophisticated options. This is coupled with a set of *novelty evaluators* for different supported feature representations (binary, multivalued, etc.), which are able to track and compute the novelty of any state in the search given any novelty feature evaluator. Implementation details are given in Section 6.3.4.
8. A built-in FSTRIPS plan validator.

6.2 Supported Features and Extensions

The FS planner supports the two basic search modes which have been described throughout this thesis. On the one hand, a greedy best-first search driven by the variations of the $h_{\max}$ and h_{FF} heuristics based on the first-order RPG (FOL-RPG), as described in Chapters 3 and 4. On the other hand, the width-based BFWS(R) family of algorithms, described in Chapter 5. In both cases, the planner is able to perform a search grounding actions or with action schemas alone, as described in Section 4.6.

Along with the core of the Functional STRIPS classical planning language, the planner supports a number of extensions which are useful both from the expressive and the computational point of view. These include existential quantification, state constraints, a library of global constraints, and the possibility of using externally-defined symbols and built-in arithmetic symbols. Due to its simplicity and to the fact that it needs to compute no heuristics, BFWS(R)-based search supports all these language features; FOL-RPG-based search support most, but not all of them.

Conditional Effects Conditional effects are fully supported in both search modes of the planner.

Bounded Integer Variables The planner supports a simple and compact mechanism to declare bounded integer types.

Externally-Defined Symbols One of the most powerful extensions of the language, as we argue in Chapters 3 and 5, is the possibility given to the modeler of providing the denotation of (fixed) function and predicate symbols p of the language through a procedure defined in some programming language. In the case of the FS planner, this procedures need to be encoded in a C++ class conforming to a particular interface.

If a search based on first-order RPG heuristics is to be used effectively in a problem with external symbols, then the arity of the symbol needs to be small enough so that the denotation can be compiled into an extensional constraint. This compilation is actually equivalent to having the modeler specify the fixed denotation *extensionally* in the problem representation (e.g. the PDDL instance file), that is, by listing all atoms of the form $p(\bar{t})$ that are true in the initial state (and in any state, given the *static* nature of the symbol). If the arity of the symbol is not small, then the modeler can provide a handcrafted constraint propagator for the symbol to be used during the construction of the first-order RPG. This, however, is time-consuming and requires familiarity with the internals of both the FS planner and Gecode.

More interestingly, BFWS-based search does not impose the requirement above, as it does not need to compute any heuristic, and only requires the ability to compute the denotation of the symbol in any given state. This means that the arity of these symbols does not need to be bound, and, in particular, that *arbitrary-arity externally-denoted symbols* can be used. These are symbols whose denotation can depend on the full state, i.e. on all of the state variables, and in that sense they can be considered as *procedural derived predicates* (see below for standard derived predicates), without having to worry about how to deal with them in the computation of heuristics, which is a non-trivial issue (Bernardini et al., 2017).

Global Constraints As described in Chapter 3, the fact that the planner interfaces with the `Gecode` constraint solver allows it to support with little effort any `Gecode` constraint, and in particular global constraint, as a predicate symbol of the language of any FSTRIPS problem, in both search modes. Currently, the planner offers support for a limited number of constraints, e.g. `alldiff`, `sum`, `nvalues`, but the addition of other constraints supported by `Gecode` requires only a few lines of code to specify their intended denotation and link them with the appropriate `Gecode` constraint. The latter is not necessary for BFWS search, where no heuristic needs to be computed, and where indeed there is no actual distinction between a global constraint-backed symbol and simply an externally-defined symbol.

Existential and Universal Quantification The planner accepts the use of existential and universal quantification in the encoding of problems. Existential quantification is dealt with efficiently in the manner described and tested in Chapter 4. Universal quantification is dealt with during preprocessing by expanding any quantifier into a conjunction of formulas over the finite universe of the quantifier type.

Universally-Quantified Effects Universal quantification in the first-order logical language $\mathcal{L}(P)$ corresponding to a FSTRIPS problem P is not to be confused with universally-quantified effects, a useful feature already present in the ADL language (Pednault, 1989). In the PACMAN domain in Fig. 5.2, for instance, we express that after a move of the Pac-Man to grid position x_1, there is a position update *for every ghost* in the problem; to do so, we use a functional effect

$$\forall\, g/\tau_G \;\; loc(g) := @\, move_G(loc(g), x_1)$$

where τ_G is the type denoting "ghost" agents, and thus the variable g is quantified over all possible ghosts in the problem. The universal quantifier here is not part of any formula whatsoever, but a feature of the problem encoding used to compactly express a large number of effects quantified over a type domain. The `FS` planner deals with universally-quantified effects by expanding them into a linear number of effects over the finite domain of the quantifier type.

Derived Predicates The planner features very rudimentary support for PDDL derived predicates, otherwise known as axioms. Derived predicates have been shown to allow for exponentially shorter problem representations in some cases (Thiébaux et al., 2005); besides, features that occur quite naturally in planning

domains such as transitive closures (e.g. the above predicate in BLOCKSWORLD), can be represented in a straight-forward manner with PDDL axioms.[1] The planner currently supports axioms with only one level of stratification.

6.3 Implementation and Optimization Details

The empirical evaluation and comparison of planners and of algorithms in general is fraught with difficulties, as witnessed by the sizeable, albeit not too conclusive, literature that addresses best practices and methodologies (McGeoch, 1996; Rardin and Uzsoy, 2001). The runtime performance of heuristic search algorithms is extremely sensitive to a plethora of implementation choices and details which often go unnoticed or, in the best case, under-reported, but can nonetheless have a decisive impact on experimental results (Junghanns and Schaeffer, 2001; Burns et al., 2012). Indeed, Hooker (1995) presents a solid case against conflating research with development and evaluating algorithms based on their running times on a set of fixed benchmarks. For better or for worse, however, many of the metrics usually employed in the planning research community depend, to some extent, on the runtime of planners, and as such, we here try to report some of those optimization and implementation details in the FS planner that through informal, preliminary testing we have become to consider as most important for the performance of the planner.

6.3.1 Grounding

Function symbols in Functional STRIPS often allow encodings where action schemas need fewer parameters than equivalent function-free representations, overall resulting in a smaller number of ground actions. On the other hand, function-free schemas usually result in much simpler, propositional ground actions, and those operations that arise more frequently during search, such as checking applicability of an action in a certain state, or computing successor states, can be implemented with specialized routines. Whether one or the other option is more performant strongly depends on the exact type of search, and eventually heuristic, that is used.

In simple problems, the FS planner can apply the *naive* grounding strategy that consists in checking, for each possible substitution of the action parameters, whether the precondition of the ground action that results from the substitution is equivalent to a contradiction or not, in which case it is assumed that might be applicable in some state. Action grounding however becomes more of a challenge when the number of potential parameter substitutions for some action schemas (i.e. the size of the Cartesian product of the domain of each action parameter) is very large (Areces et al., 2014). This is the case for many of the standard benchmarks used in the community. To name a couple, the Sokoban domain from the 2011 International Planning Competition, or the Tetris domain from the last edition of the competition, in 2014, feature action schemas with up to 7 parameters, and the number of potential parameter substitutions in the largest instances is in the order of the billions. Other problems that arise naturally in other domains such as organic synthesis (Matloob

[1] PDDL axioms are slightly different than axioms as originally defined in STRIPS (Lifschitz, 1987); the former are actually stratified, function-free logic programs, while the latter are first-order logic axioms. It is well known for instance that first order logic cannot express transitive closures (Papadimitriou, 1994), while logic programs can.

and Soutchanski, 2016), can be readily cast as planning problems, but the number of potential parameter substitutions is also in the order of the billions.

In these cases, the naive grounding strategy is not a sensible option. The `FS` planner implements, thanks to work by Nathan Robinson and Miquel Ramírez, a more sophisticated grounding module that makes use of a compilation to a logic program in order to perform reachability analysis and grounding all in one, roughly as described in (Helmert, 2009, Section 6). This compilation is particular to both the set of action schemas and the initial state of the search. The reachability analysis is often able to infer that certain ground actions will never be applicable when starting from that initial state, and that some fluents will never actually change their value, for which they can actually be considered as static facts, not fluents. This type of analysis not only results in more powerful pruning of actions and fluents, but in most cases is orders of magnitude more efficient than the naive strategy, often performing the grounding in less than one second, even for problems with number of ground actions in the tens of thousands. This logic programming grounding currently only works when no functions, externally-defined symbols or global constraints are used. In contrast to (Helmert, 2009), we do not directly solve the generated logic program but use off-the-shelf the `Clingo` ASP solver (Gebser et al., 2012) to that end.

6.3.2 State Representation

The representation of the state is well known to be a key implementation detail in heuristic search in general, and in heuristic planning in particular (Edelkamp and Helmert, 1999; Helmert, 2009; Burns et al., 2012). The `FS` planner uses a hybrid state representation where the representation of any state contains both a bitmap (a `C++` vector of `bools`), to encode the value of all binary state variables, and a more general data structure (namely, `C++` vector of `ints`), to encode the value of all multivalued state variables. State variables are indexed according to their type, and the access to the value of any such variable in any state requires an additional indirection to determine in which of the two data structures the actual value needs to be looked up. This arrangement is *not* optimal for fully-propositional planning nor for fully-multivalued planning, as it imposes in both cases an unnecessary overhead, but it strikes a reasonable balance between performance, on the one hand, and generality of the code and ease of maintainability, on the other.

6.3.3 Optimization of the First-Order Relaxed Planning Graph

The computation of the first-order relaxed planning graph is mapped by the `FS` planner into a series of constraint satisfaction problems. Chapter 3 gives a high-level description of the form of the $\Gamma(\phi, P_k)$ CSP that is constructed by the planner to check whether a certain first-order formula ϕ is satisfied in layer P_k of this relaxed planning graph. We here briefly review a number of alternative CSP models and optimizations supported by the planner.

Incremental Computation of Layers The current implementation exploits the monotonicity of the relaxed planning graph to speed up CSP resolution from layer to layer. To this end, an extra constraint is added to the $\Gamma(\phi, P_k)$ constraint satisfaction problem to enforce the solutions of the CSP to result in *novel* values for the layer domains. To illustrate, assume that we have an action effect $x = x + 1$, where $\underline{x}$ is a state variable, and we want to compute the

values that this effect supports in layer P_k of the graph. Assume also that the values that are reachable for x in the previous layer P_{k-1} are $x^{k-1} = [0, 10]$, the domain of x is $[1, 100]$ and the action precondition is, accordingly, $x < 100$. As per the discussion in Chapter 3, the CSP that will be generated will have a single CSP variable v_x, with domain $D(v_x) = [1, 100]$, a constraint $v_x < 100$ given by the precondition of the action, and a constraint $v_x \in [0, 10]$ given by the previous layer of the relaxed graph. All the solutions to this CSP will be exhaustively explored, which amounts to 11 different solutions, and the effect will be evaluated for each of them, to conclude that the values supported in layer P_k are $[1, 11]$. Of these values, of course, the first 10 were already supported, so there was little need to go through all of them.

To avoid having to iterate through the CSP solutions corresponding to value that are already reachable, an extra variable v_{x+1} can be used to model the value taken by the term $x+1$. To keep the semantics of the effect, a constraint $v_{x+1} = v_x + 1$ needs to be added as well, and then a constraint $v_{x+1} \notin [0, 10]$, to ensure that all CSP solutions *lead to novel values* for state variable x. The above example is simple, but its generalization to arbitrary actions is straightforward, except when the left-hand side of the effects involves nested fluents, in which case the optimization is not applied, as it is not possible to know in advance which is the actual state variable that will be affected by the effect.

One CSP per ground action A very simple alternative to posting one CSP to check the values supported by every *effect* in any RPG layer is posting one CSP per every *action*. This makes the incremental computation of layers described above slightly more involved, but still feasible, and usually pays off.

One CSP per action schema As discussed in Chapter 4, a possible way of tackling an excessive number of ground actions *during the computation of the first-order RPG* is to compute the graph with the action schemas instead of the ground actions. This adds at least as many additional CSP variables as parameters has the action schema, and in effect means that the parameter substitution of the actual ground action that supports novel values is computed *as part of the CSP solution*. The resulting CSPs are usually more complex, but the (potentially exponential) reduction in the number of CSPs that need to be dealt with sometimes compensates for the overhead. An interesting middle ground is to do what we call *selective grounding*: have the grounding applied at preprocessing substitute only *some* of the action parameters, and compute the first-order RPG with half-grounded actions. If this is done intelligently to ground only those parameters which actually make the CSP more complex (not all of them do), then the result can potentially combines the benefits of grounded-action RPGs (simple, quick-to-solve CSPs) with the benefits of action-schema CSPs (less CSPs to be solved).

6.3.4 Optimization of Width-Based Algorithms

The BFWS(R) family of algorithms presented in Chapter 5 has an unusual performance profile, in that there is no need to compute any heuristic during the search. In first-order RPG-based search algorithms presented in Chapters 3 and 4, for instance, the computation of the heuristic can typically take up to 90% of all the runtime of the planner. In the case of BFWS(R), the node generation rate is comparatively much

higher, and the bottleneck of the computation usually lies in checking action applicability and generating successor states. The computation of the state novelty values appears to be less time-consuming than the previous, although this largely varies depending on the characteristics of the problem, particularly number of ground actions and of state variables. A systematic analysis of the relative costs of each component of the search remains as future work.

The planner features optimizations addressing both parts of the search, state transitions and novelty computations. As far as the former is concerned, when the number of ground actions is large enough, the set $A(s)$ of actions applicable in any state s of the search is computed and represented through a *successor generator*-like strategy as described in (Helmert, 2006a, Section 5.3). The implementation of this successor generator is geared towards propositional STRIPS problems, where its positive impact is most visible. Its use is not restricted to width-based algorithms, indeed, it can be used in any search mode supported by the planner.

As far as novelty computations are concerned, there are a couple of optimizations worth mentioning. First, following Lipovetzky and Geffner (2017a), the planner only computes novelty-1 and novelty-2 measures; novelty-3 measures and beyond are usually too expensive. This means that all nodes with novelty larger than 2 are considered to have novelty 3 as far as their priority in the greedy best-first search is concerned. When the number of atoms is too large to fit in memory the data structures that are necessary to compute novelty efficiently, even novelty-2 computations are skipped. Novelty-k tables keep track of the atoms ($k = 1$) and atom pairs ($k = 2$) that have so far been reached during the search. Different implementations of these tables and of the novelty evaluators coexist in the planner codebase, and are selected for use depending on whether (a) novelty features other than the value of state variables are used or not, and (b) the number of possible atoms $x = v$ is small enough. When only state-variable features are used, the computation of the novelty of a state can be optimized to directly use the actual state representation instead of some intermediate representation of feature values; when the number of possible atoms is relatively small, all possible atoms, and even pairs of atoms, can be assigned an integer index, in which case a novelty-k table, for $k \in \{1, 2\}$, can be implemented with a bitmap and requires $|F|^k$ bits. When novelty $w_{u,r}(s)$ given values $u(s)$ and $r(s)$ is used, as in the best-performing algorithms described in Chapter 5, this raises to $|G| \times |R| \times |F|^k$ bits, where G is the set of goal conditions and R the set of relevant atoms. In standard benchmarks such as Transport, Tetris or Parking, where the number of atoms is in the thousands, this might prove too large for $k = 2$, hence the skipping of novelty-2 computations that we mentioned above.

Additionally, the fact that the BFWS(R) works with no knowledge of the declarative representation of actions imposes a certain penalty to the computation of the novelty of any state. Indeed, when the action representation is available, testing if the state generated by an action has novelty 1 can be done in constant time by checking whether each one of the (typically bounded number of) atoms added by the action is new. When that action representation is not available, however, this test is linear in the number of variables, as there is no information about action effects, and the value of all variables in the resulting state needs to be checked. In general, information on action representation allows novelty-i tests to be performed in time exponential in $i - 1$; when that information is lacking, the required time is exponential in i.

For this and other optimization reasons, the planner adopts a *lazy* approach to compute novelty values and even the set R of relevant atoms. Each time a state is generated, the planner only computes whether the state has novelty 1 or greater than 1. Novelty measures greater than 1 are computed only when no node with novelty 1 remains in the open list of the search. Additionally, before computing the set R and the actual $w_{u,r}$ novelty measures, nodes are evaluated using novelty w_u measures only. This can progress the search, and even solve some problems, without the potentially expensive computation of R on the initial state. Our implementation of this lazy search uses a schema with multiple open lists, each with a different priority. Hence, there is for instance a queue for nodes n such that $w_u(n) = 1$. Only when nodes in that queue are exhausted, a second queue is explored that contains all nodes n that *potentially* might have novelty $w_{u,r}(n) = 1$, and only then the actual set R is computed from the initial state s_0, so that $w_{u,r}$ computations can be performed. Some problems greatly benefit from this lazy approach: Visitall problems, for instance, can be directly solved by considering only nodes from the first queue, i.e. with novelty w_u equal to one.

Chapter 7

Conclusions

> Traducir de una lengua en otra [...] es como quien mira los tapices flamencos por el revés, que aunque se veen las figuras, son llenas de hilos que las escurecen y no se veen con la lisura y tez de la haz.
>
> ⋆⋆⋆
>
> Translating from one language to another [...] is like looking at Flemish tapestries from the wrong side, for although the figures are visible, they are covered by threads that obscure them, and cannot be seen with the smoothness and color of the right side.
>
> Miguel de Cervantes Saavedra, *El ingenioso hidalgo don Quijote de la Mancha*

Let us conclude this thesis with a summary of our contributions and a discussion of potential lines of research that might follow this work, including some which are already being explored.

7.1 Summary of Contributions

This thesis provides two main contributions linked to the same underlying *leitmotiv*: exploring effective means of dealing with expressive modeling languages in planning.

The first of these contributions is the analytical and empirical evidence that the use of more expressive modeling languages can be beneficial both for modeling and computation. In spite of the well-known fact that more expressive languages are *worst-case* harder to reason with (Levesque and Brachman, 1985), we have argued and illustrated with several examples that in order to solve a certain, fixed problem, it might be more convenient to represent it with the more expressive language, which can capture relevant structure of the problem to be exploited computationally. We have performed a logical analysis of the relaxed planning graph (RPG) that has resulted in a stronger inference mechanism, which we have named *first-order relaxed*

planning graph. This construct is based on the standard notion of first-order logic interpretation, is simple, elegant, and supports *function symbols* (Chapter 3) and *existential quantification* (Chapter 4) in the definition of the problem actions and goal. We have shown how to map the computation of the first-order RPG into a series of constraint satisfaction problems that can be solved off-the-shelf with a CSP solver. The heuristics derived from the first-order RPG are more informed than their standard counterparts, although they are no longer worst-case tractable. In practice, the form of action descriptions and goal formulas is often simple enough so that this is not a problem; when this is not the case, they can be (polynomially) approximated thanks to standard constraint propagation techniques.

The use of a constraint solver opens up the possibility of using some other language extensions such as global constraints in the definition of the problem. We have shown that these extensions are convenient not only from the expressive but also from the computational point of view. The efficient support for existential quantifiers in action preconditions and goal formulas that our heuristics provide is interesting because many planning problems can be modeled in a natural way by making use of them, but also because, as we have shown, existential quantification opens up an interesting connection between planning and constraint satisfaction, as any constraint satisfaction problem can be expressed as a action-less, fluent-less planning problem. We also show how the same mechanism that allows us to support existential quantification can be used to search for plans and to compute heuristics without having to ground the (lifted) action schemas in the problem description.

The second of the contributions (Chapter 5) is the development of a novel algorithm that, unlike most of the standard classical planning techniques (planning as satisfiability, heuristic search planning, partial order planning, etc.), is able to plan *efficiently* without a declarative representation of the problem actions and almost without a declarative representation of the problem goal. This is an original contribution that represents a significant departure from mainstream planning research over the last decades. Our algorithm is based on notions of *state novelty* and *width-based* search recently proposed by Lipovetzky and Geffner (2012, 2017a), and only requires information about the structure of the states of the problem and a count of the number of subgoals that have been reached in any state, plus a black-box implementation of the transition function. In spite of the handicap of not being able to reason on the structure of actions, we show with an extensive experimental study that the algorithm is competitive with the state of the art methods in benchmarks from the last two international planning competitions. We do not take this as suggesting that modeling languages are not important; on the contrary, we think this provides a basis from which expressive modeling features can be given adequate support. We illustrate this by discussing a few problems such as (deterministic versions of) pacman or pong. These are clearly classical planning problems, in that they have an omniscient agent and actions with deterministic effects, but nonetheless seem to be beyond the expressive capabilities of languages such as PDDL. The combination of function symbols and, crucially, of logical symbols with denotations provided by external procedures, is key to representing this problems compactly. These features are all seamlessly supported by our width-based algorithms and the `FS` planner.[1]

[1] The `FS` planner, described in detail in Chapter 6, is open-sourced and freely available at `https://github.com/aig-upf/fs`.

7.2 Ongoing and Future Work

The work on this thesis suggests a number of lines of research which we think might be interesting. We briefly review some of them here, in no particular order.

Automatic Problem Reformulation. The results presented in this thesis constitute significant evidence that offering the modeler an expressive language to represent whatever problem needs to be solved might be a win-win situation. On the one hand, the modeler can avoid unnecessary workarounds and produce a representation which is more readable, maintainable, and elaboration tolerant (McCarthy, 1998). On the other hand, and from the point of view of the designer of a planner, there are different possibilities to deal with the problem. One is to extend the mechanisms by which the planner operates (heuristic search, satisfiability compilations, etc.) so that they can cope with the more expressive language features. This is a time-consuming task, but as we have seen in this thesis, it may pay off computationally. Indeed, some recent propositional planners do recognize the computational value of these constraints, and try to *recover* them from the low-level representations. An example of this would be *at-most-one* constraints in purely relational representations. For any block b in the blocks-world, for instance, *at most one* of the atoms $at(b, c)$, $at(b, d)$, $at(b, e)$, etc. can be true at the same time (Edelkamp and Helmert, 1999; Helmert, 2009). Another example would be the use of logical *axioms*, which in some situations allows not only for exponentially smaller representations, but also smaller search spaces (Thiébaux et al., 2005; Ivankovic and Haslum, 2015). Miura and Fukunaga (2017) have recently shown some possible benefits of automatically inferring simple forms of axioms. In general, recovering this type of structural information to leverage it computationally is not a trivial process, and in many cases the effort might be spared if only the modeler would have had a higher-level language at hand in the first place, allowing her to represent that information directly.

Another option for the planner designer would be to use *automatic reformulation* techniques to compile the high-level representation into a lower-level equivalent, something which one would expect to be easier than the compilation in the inverse direction. In the case of the language features discussed in this thesis, we have already mentioned how functions can be compiled into existential variables by following the principles of Skolemization in reverse, as described in Haslum (2008); other methods might of course produce more compact compiled representations. In turn, existential variables can be compiled away, as we have discussed in Chapter 4. All these transformations produce problem representations which are *equivalent* in the sense that solutions of both source and target representations can be polynomially mapped into one another, but they are not necessarily equivalent from the computational standpoint, as we have argued.

Our point is that there is no gain in forcing the problem modeler use a low level language: if the planner can deal with high-level constructs, so much the better; if not, these constructs can be automatically translated into lower-level equivalents, hopefully in a way that benefits the particular techniques used by the planner at hand (Haslum, 2007). Incidentally, this approach has been explored to considerable success in the field of constraint satisfaction, as in e.g. the Zinc family of expressive languages and the automatic transformations between them (Nethercote et al., 2007; Marriott et al., 2008).

Planning with Data Types. The constraint satisfaction techniques that we saw (Chapters 3 and 4) can help build a first-order relaxed planning graph can be applied more or less off-the-shelf not only to integer variables, but also to structured data types such as sets or intervals (Gervet, 1997). Constraint programming frameworks such as Gecode do indeed support reasoning over such variable types, and extending our heuristics to support them too should not be a challenge, and would in many cases significantly ease the task of modeling. The width-based planning algorithm presented in Chapter 5 would be even more powerful in that regard. Our algorithm should be able to to support planning over any background theory with little additional effort, as long as the basic semantics of the theory is provided to implement the (black-box) transition function of the planning model that the algorithm requires. This could be seen as a straight-forward implementation roadmap for the *planning modulo theories* paradigm (Gregory et al., 2012), which has yet to fulfill its enormous potential.

Lifted Planning. On an unrelated note, the prospect of a planner which is able to plan effectively by directly using the lifted action schemas of the problem (i.e. no grounding involved), which we have described in Chapter 4, also suggests interesting possibilities. Indeed, the classical planning benchmarks used in standard planning competitions already include some benchmarks where the huge number of ground actions is problematic for those approaches which ground all of the lifted actions at preprocessing. Recent research has shown that a similar thing happens with some meaningful problems from the natural sciences that can be readily cast as planning problems. These include for instance the computation of the genome edit distances used to assess the plausibility of different evolutionary trees or the synthesis of molecules by means of appropriate sequences of chemical reactions (Haslum, 2011; Matloob and Soutchanski, 2016).[2] The intuitive encodings for both of these examples contain action schemas with large signatures that are beyond the possibilities of most current planners. In the case of the organic synthesis problem, for instance, this is not because the problem has some particularly challenging combinatorial structure, but because the size of the ground representation is orders of magnitudes larger than the standard problems in the benchmarks, thus invalidating the apparently innocuous assumption that grounding the problem representation at preprocessing simplifies the subsequent reasoning. We are already working on the application of lifted planning techniques as described in Section 4.6 to this type of problems.

Hybrid Planning. The `FS` planner developed in this thesis has recently been transformed by Ramirez et al. (2017) into the `FS+` planner, which extends our first order relaxed planning graph to cope with *hybrid planning problems*, i.e. planning problems with real-valued state variables and both continuous and instantaneous change. `FS+` precisely supports the PDDL+ language extended with function symbols, including arithmetic expressions, arbitrary algebraic and trigonometric functions and *autonomous processes* (Fox and Long, 2002, 2003, 2006; Cashmore et al., 2016; Scala et al., 2016a). Work to merge both the `FS+` and `FS` planners is currently undergoing, including the exploration of meaningful generalizations to real-valued variables of the notions of state novelty on which the work presented in Chapter 5 is based.

[2] The first of these problems was already part of the 2014 international planning competition.

Learning Novelty Features for Width-Based Search. An issue that we only briefly mentioned in Chapter 5 related to our width-based search algorithm is the use of what we have called *ranked feature lists* to prioritize search nodes during the search. This mechanism indeed opens up a novel way of incorporating domain knowledge (Bacchus and Kabanza, 2000; Haslum and Scholz, 2003) in the form of *novelty features*, i.e. arbitrary first-order formulas which are used to compute the novelty of a given search node. These features can be specified by the modeler, but machine learning techniques could also conceivably be used to infer from training samples which features work best. Some of the preliminary results that we have obtained on this are encouraging, showing significant search speed-ups, but the challenge remains to gain a systematic understanding of the kind of features that can be beneficial in this type of width-based search algorithms.

Appendix A

First-Order Logic

In this appendix we provide a brief overview of the first-order logic background necessary to follow the formalization of the Functional STRIPS classical planning language and the discussion on the First-Order Relaxed Planning Graph presented in Chapters 2 to 4. A more systematic account can be found in any standard textbook (Enderton, 2001; Rautenberg, 2006). Our exposition here follows the formalization by Reiter (2001).

We begin by presenting the syntax and semantics of first-order languages with *types* (or *sorts*), usually called *many-sorted languages*. It is known that many-sorted logics are no more expressive than standard unsorted logics: any typed first-order sentence can be rewritten into a type-free sentence with some auxiliary predicates, and such transformation preserves the notion of satisfiability (Reiter, 2001). We will however often work with many-sorted definitions in the rest of this thesis, so we prefer to give a direct formalization of them.

Definition A.1 (Many-Sorted First-Order Logic)**.** *A many-sorted first-order* language $\mathcal{L}$ *is made up of:*

- *A non-empty set* T *of* types *or* sorts*.*
- *An infinite number of variables* $x_1^\tau, x_2^\tau, \ldots$ *for each type* $\tau \in T$*.*
- *For each* $n \geq 0$ *and each tuple* $\langle \tau_1, \ldots, \tau_{n+1} \rangle \in T^{n+1}$ *of types, a (possibly empty) set of* function symbols*, each of which is said to have* arity n *and* type $\langle \tau_1, \ldots, \tau_{n+1} \rangle$*.*
- *For each* $n \geq 0$ *and each tuple* $\langle \tau_1, \ldots, \tau_n \rangle \in T^n$ *of types, a (possibly empty) set of* relation symbols*, each of which is said to have* arity n *and* type $\langle \tau_1, \ldots, \tau_n \rangle$*.*

Nullary (i.e. arity-0) functions are called *constants*. We will usually say that a constant with type $\langle \tau \rangle$ has simply type τ. We will also omit the superscript in x^τ when it is clear from the context the type of variable x. Terms and formulas in the language can be defined inductively as follows:

Definition A.2 (First-Order Terms)**.**

- *Any variable* x^τ *of the language is a term with type* τ*.*
- *Any constant symbol of the language with type* τ *is a term with the same type.*

- *If $t_1, \ldots, t_n$ are terms with respective types $\tau_1, \ldots, \tau_n$, and f is a function symbol with type $\langle \tau_1, \ldots, \tau_n, \tau_{n+1} \rangle$, then $f(t_1, \ldots, t_n)$ is a term with type τ_{n+1}.*

It is clear from the above that any term t in the language can be assigned a unique type, which we will denote by $type(t)$.

Definition A.3 (First-Order Formulas)**.**

- *If t_1 and t_2 are two terms with the same type, then $t_1 = t_2$ is an (*atomic*) formula.*
- *If $t_1, \ldots, t_n$ are terms with respective types $\tau_1, \ldots, \tau_n$, and R is a relation symbol with type $\langle \tau_1, \ldots, \tau_n \rangle$, then $R(t_1, \ldots, t_n)$ is an (atomic) formula.*
- *If ϕ_1 and ϕ_2 are formulas, then $\neg\phi_1$, $\phi_1 \vee \phi_2$ and $\phi_1 \wedge \phi_2$ are also formulas.*
- *If ϕ is a formula, then $\exists_t x^\tau \phi$ and $\forall_t x^\tau \phi$ are also formulas.*

Quantification happens over a certain type, i.e. for each type $\tau \in T$ there are universal and existential quantifier symbols $\forall_\tau$ and $\exists_\tau$, which must be applied to variables of the same type. We sometimes abbreviate with $\exists\ x_1/\tau_1, \ldots, x_n/\tau_n$ the actual quantification prefix $\exists_{\tau_1} x_1^{\tau_1} \ldots \exists_{\tau_n} x_1^{\tau_n}$, and likewise for universal quantification. We will omit type information when the context is clear enough. A formula with no existential ($\exists$) or universal ($\forall$) quantifier, we will call *quantifier-free*.

As customary, we will use $\phi_1 \to \phi_2$ as shorthand for $\neg\phi_1 \vee \phi_2$ and $\phi_1 \leftrightarrow \phi_2$ as shorthand for $(\phi_1 \to \phi_2) \wedge (\phi_2 \to \phi_1)$. We will sometimes assume that the symbols $\top$ (tautology) and $\bot$ (contradiction) are also atomic formulas.

Intuitively, a variable x is said to be *free* if it does not occur in the *scope* of any quantification prefix $\exists x$ or $\forall x$, and the variable is bound otherwise. If $\bar{x} = \langle x_1, \ldots, x_n \rangle$ is a tuple with all free variables in a formula ϕ, we sometimes denote the formula by $\phi[\bar{x}]$. A formula without free variables is called a *sentence*.

We can now formalize the semantics of a first-order language, based on the notion of *model*.

Definition A.4 (First-order logic model)**.** *Let $\mathcal{L}$ be a many-sorted first-order language. A first-order* model *(sometimes called* structure *or* interpretation*) for the language $\mathcal{L}$ is a tuple*

$$\mathcal{M} = \langle \{\mathcal{U}_\tau\}, \{f^{\mathcal{M}}\}, \{P^{\mathcal{M}}\} \rangle$$

made up of:

- *A non-empty set $\mathcal{U}_\tau$, the* universe *of type τ, for each $\tau \in T$. We often assume that the universes of all types are disjoint and use $\mathcal{U} = \bigcup_{\tau \in T} \mathcal{U}_t$ for brevity.*
- *For each n-ary function symbol f of the language with type $\langle \tau_1, \ldots, \tau_n, \tau_{n+1} \rangle$, a function $f^{\mathcal{M}} : \mathcal{U}_{\tau_1} \times \cdots \times \mathcal{U}_{\tau_n} \to \mathcal{U}_{\tau_{n+1}}$. When $n = 0$ and f is a constant symbol of type τ, $f^{\mathcal{M}}$ is simply some element of the universe $\mathcal{U}_\tau$.*
- *For each n-ary predicate symbol P of type $\langle \tau_1, \ldots, \tau_n \rangle$, a subset $P^{\mathcal{M}} \subseteq \mathcal{U}_{t_1} \times \cdots \times \mathcal{U}_{\tau_n}$. If $n = 0$, we will assume that $P^{\mathcal{M}}$ is a truth value $P^{\mathcal{M}} \in \{\top, \bot\}$.*

The notion of denotation of a term and truth of a formula under a given interpretation requires that we take into account all possible free variables occurring in the term of formula. We will do that by extending interpretations with a type-consistent assignment of values to free-variables. Assume that $\phi[\bar{x}]$ is a formula in some first-order language $\mathcal{L}$, $\mathcal{M}$ an interpretation for that language, and σ a *variable assignment* for $\bar{x}$, i.e. a function mapping any free variable x^τ in the tuple $\bar{x}$ to an element in $\mathcal{U}_\tau$.

The assignment σ can be easily extended into a function σ^* that gives the denotation of any term in the language, defined as follows:

1. For any variable x, $\sigma^*(x) = \sigma(x)$.
2. For terms $t_1, \ldots, t_n$ and a n-ary function symbol f with matching type,

$$\sigma^*(f(t_1, \ldots, t_n)) = f^{\mathcal{M}}(\sigma^*(t_1), \ldots, \sigma^*(t_n)).$$

We say that ϕ *is true under interpretation* $\mathcal{M}$, when its free variables are given values according to assignment σ, denoted by $\mathcal{M} \vDash \phi(\sigma)$, in the following cases:

- For any two terms t_1, t_2, $\mathcal{M} \vDash (t1 = t2)(\sigma)$ iff $\sigma^*(t_1)$ and $\sigma^*(t_1)$ are the same element.
- For any n-ary predicate symbol P and terms $t_1, \ldots, t_n$ of appropriate type, $\mathcal{M} \vDash P(t_1, \ldots, t_n)(\sigma)$ iff $\langle \sigma^*(t_1), \ldots, \sigma^*(t_n) \rangle \in P^{\mathcal{M}}$.
- $\mathcal{M} \vDash (\neg\phi)(\sigma)$ iff not $\mathcal{M} \vDash \phi$.
- If $\phi \equiv \phi_1 \wedge \phi_2$, then $\mathcal{M} \vDash \phi(\sigma)$ iff $\mathcal{M} \vDash \phi_1$ and $\mathcal{M} \vDash \phi_2$.
- If $\phi \equiv \phi_1 \vee \phi_2$, then $\mathcal{M} \vDash \phi(\sigma)$ iff $\mathcal{M} \vDash \phi_1$ or $\mathcal{M} \vDash \phi_2$.
- $\mathcal{M} \vDash (\exists_\tau x\ \phi)(\sigma)$ iff $\mathcal{M} \vDash \phi(\sigma[x/a])$, for some $a \in \mathcal{U}_\tau$.
- $\mathcal{M} \vDash (\forall_\tau x\ \phi)(\sigma)$ iff $\mathcal{M} \vDash \phi(\sigma[x/a])$ for every $a \in \mathcal{U}_\tau$.

In the above definition, $\sigma[x/a]$ is the function that assigns values as in σ except to variable x, which is assigned value a.

Definition A.5 (Satisfaction and Validity)**.** *Let ϕ be a formula in some first-order language $\mathcal{L}$. We say that*

- *An interpretation $\mathcal{M}$* satisfies *ϕ iff $\mathcal{M} \vDash \phi(\sigma)$ for any possible assignment σ. ϕ is* satisfiable *iff there is some interpretation $\mathcal{M}$ that satisfies it.*
- *ϕ is a* valid *formula, denoted by $\vDash \phi$, iff every possible first-order interpretation of the language $\mathcal{L}$ satisfies ϕ.*

Determining if a first-order logic sentence is valid is an undecidable problem, and so is determining if it is satisfiable.

Bibliography

Aldinger, J., Mattmüller, R., and Göbelbecker, M. Complexity of interval relaxed numeric planning. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 19–31. Springer, 2015.

Alviano, M., Faber, W., and Leone, N. Disjunctive ASP with functions: Decidable queries and effective computation. *Theory and Practice of Logic Programming*, 10 (4-6):497–512, 2010.

Amarel, S. On representations of problems of reasoning about actions. *Machine intelligence*, 3(3):131–171, 1968.

Appiah, K. A. *Thinking it through: An introduction to contemporary philosophy*. Oxford University Press, 2003.

Areces, C., Bustos, F., Dominguez, M., and Hoffmann, J. Optimizing planning domains by automatic action schema splitting. In *International Conference on Automated Planning and Scheduling*, pages 11–19. AAAI Press, 2014.

Bacchus, F. The AIPS'00 planning competition. *AI Magazine*, 22(3):47–56, 2001.

Bacchus, F. The power of modeling - a response to PDDL 2.1. *Journal of Artificial Intelligence Research*, 20:125–132, 2003.

Bacchus, F. GAC via unit propagation. In *International Conference on Principles and Practice of Constraint Programming*, pages 133–147. Springer, 2007.

Bacchus, F. and Kabanza, F. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

Bäckström, C. Planning using transformation between equivalent formalisms: A case study of efficiency. In *AAAI Workshop on Comparative Analysis of AI Planning Systems*, 1994.

Bäckström, C. Expressive equivalence of planning formalisms. *Artificial Intelligence*, 76(1):17–34, 1995.

Backstrom, C. and Jonsson, P. All PSPACE-complete planning problems are equal but some are more equal than others. In *Annual Symposium on Combinatorial Search*, pages 10–17, 2011.

Bäckström, C. and Klein, I. Planning in polynomial time: the SAS-PUBS class. *Computational Intelligence*, 7(3):181–197, 1991.

Bäckström, C. and Nebel, B. Complexity results for SAS+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

Bäckström, C., Chen, Y., Jonsson, P., Ordyniak, S., and Szeider, S. The complexity of planning revisited: a parameterized analysis. In *AAAI Conference on Artificial Intelligence*, pages 1735–1741, 2012.

Bäckström, C., Jonsson, P., Ordyniak, S., and Szeider, S. A complete parameterized complexity analysis of bounded planning. *Journal of Computer and System Sciences*, 81(7):1311–1332, 2015.

Baral, C., Gelfond, M., and Provetti, A. Representing actions: Laws, observations and hypotheses. *The Journal of Logic Programming*, 31(1-3):201–243, 1997.

Barták, R. and Vodrážka, J. The effect of domain modeling on efficiency of planning: Lessons from the Nomystery domain. In *Conference on Technologies and Applications of Artificial Intelligence*, pages 433–440. IEEE, 2015.

Beldiceanu, N., Carlsson, M., and Rampon, J.-X. Global constraint catalog. http://sofdem.github.io/gccat/, 2012.

Bellemare, M., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Bernardini, S., Fox, M., Long, D., and Piacentini, C. Boosting search guidance in problems with semantic attachments. In *International Conference on Automated Planning and Scheduling*, pages 29–37. AAAI Press, 2017.

Bessiere, C. and Van Hentenryck, P. To be or not to be... a global constraint. In *Principles and Practice of Constraint Programming*, pages 789–794. Springer, 2003.

Betz, C. and Helmert, M. Planning with h^+ in theory and practice. In *Annual Conference on Artificial Intelligence*, pages 9–16. Springer, 2009.

Biere, A., Heule, M., and van Maaren, H. *Handbook of satisfiability*. IOS press, 2009.

Blum, A. and Furst, M. Fast planning through planning graph analysis. In *International Joint Conference on Artificial Intelligence*, pages 1636–1642. Morgan Kaufmann, 1995.

Boddy, M. Imperfect match: PDDL 2.1 and real applications. *Journal of Artificial Intelligence Research*, 20:133–137, 2003.

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

Bonet, B., Loerincs, G., and Geffner, H. A robust and fast action selection mechanism for planning. In *AAAI Conference on Artificial Intelligence*, pages 714–719, 1997.

Bonet, B. and Geffner, H. HSP: Heuristic search planner. In *The AIPS-98 Planning Competition*, 1998.

Bonet, B. and Geffner, H. Planning as heuristic search: New results. In *European Conference on Planning*, pages 360–372. Springer, 1999.

Bonet, B. and Geffner, H. Planning as heuristic search. *Artificial Intelligence*, 129 (1–2):5–33, 2001b.

Bonet, B. and Helmert, M. Strengthening landmark heuristics via hitting sets. In *European Conference on Artificial Intelligence*, pages 329–334, 2010.

Bonet, B. and van den Briel, M. Flow-based heuristics for optimal planning: Landmarks and merges. In *International Conference on Automated Planning and*

Scheduling, pages 47–55. AAAI Press, 2014.

Boroditsky, L. How language shapes thought. *Scientific American*, 304(2):62–65, 2011.

Brewka, G., Eiter, T., and Truszczyński, M. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

Brooks, R. Elephants don't play chess. *Robotics and autonomous systems*, 6(1-2): 3–15, 1990.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Bryce, D. and Kambhampati, S. A tutorial on planning graph based reachability heuristics. *AI Magazine*, 28(1):47, 2007.

Burns, E., Hatem, M., Leighton, M., and Ruml, W. Implementing fast heuristic search code. In *Fifth Annual Symposium on Combinatorial Search*, pages 25–32, 2012.

Burt, C., Lipovetzky, N., Pearce, A., and Stuckey, P. Scheduling with fixed maintenance, shared resources and nonlinear feedrate constraints: A mine planning case study. In *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR)*, pages 91–107, 2015.

Bylander, T. The computational complexity of STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.

Cashmore, M., Fox, M., Long, D., and Magazzeni, D. A compilation of the full PDDL+ language into SMT. In *AAAI Workshop on Planning for Hybrid Systems*, 2016.

Cenamor, I., De La Rosa, T., and Fernández, F. IBaCoP and IBaCoP2 planner. *IPC 2014 planner abstracts*, pages 35–38, 2014.

Chapman, D. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–377, 1987.

Chen, H. and Giménez, O. Causal graphs and structurally restricted planning. *Journal of Computer and System Sciences*, 76(7):579–592, 2010.

Chenoweth, S. V. On the NP-hardness of Blocks world. In *AAAI Conference on Artificial Intelligence*, pages 623–628, 1991.

Coles, A., Coles, A., Olaya, A. G., Jiménez, S., López, C. L., Sanner, S., and Yoon, S. A survey of the seventh international planning competition. *AI Magazine*, 33 (1):83–88, 2012.

Coles, A., Fox, M., Long, D., and Smith, A. A hybrid relaxed planning graph-LP heuristic for numeric planning domains. In *International Conference on Automated Planning and Scheduling*, pages 52–59. AAAI Press, 2008.

Cook, S. A. The complexity of theorem-proving procedures. In *Annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

Dal Lago, U., Pistore, M., and Traverso, P. Planning with a language for extended goals. In *AAAI Conference on Artificial Intelligence*, pages 447–454, 2002.

Davies, T., Pearce, A., Stuckey, P., and Lipovetzky, N. Sequencing operator counts.

In *International Conference on Automated Planning and Scheduling*, pages 61–69. AAAI Press, 2015.

Davis, M. and Putnam, H. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

De Giacomo, G., Lespérance, Y., and Levesque, H. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121 (1-2):109–169, 2000.

Dechter, R. *Constraint Processing*. Morgan Kaufmann, 2003.

Descartes, R. *Discourse on Method. For Conducting Reason and Seeking the Truth in the Sciences*. Yale University Press, 1996. Original work published in 1637.

Do, M. B. and Kambhampati, S. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132(2):151–182, 2001.

Domshlak, C. and Nazarenko, A. The complexity of optimal monotonic planning: the bad, the good, and the causal graph. *Journal of Artificial Intelligence Research*, 48:783–812, 2013.

Domshlak, C., Helmert, M., Karpas, E., Keyder, E., Richter, S., Röger, G., Seipp, J., and Westphal, M. BJOLP: The big joint optimal landmarks planner. In *The 2011 International Planning Competition Booklet*, pages 91–95, 2011.

Domshlak, C., Hoffmann, J., and Katz, M. Red–black planning: A new systematic approach to partial delete relaxation. *Artificial Intelligence*, 221:73–114, 2015.

Dornhege, C., Eyerich, P., Keller, T., Trüg, S., Brenner, M., and Nebel, B. Semantic attachments for domain-independent planning systems. In *International Conference on Automated Planning and Scheduling*, pages 114–121. AAAI Press, 2009.

Dornhege, C., Eyerich, P., Keller, T., Trüg, S., Brenner, M., and Nebel, B. Semantic attachments for domain-independent planning systems. In *Towards service robots for everyday environments*, pages 99–115. Springer, 2012.

Doyle, J. and Patil, R. Two theses of knowledge representation: Language restrictions, taxonomic classification, and the utility of representation services. *Artificial intelligence*, 48(3):261–297, 1991.

Dreyfus, H. *What computers can't do: The limits of artificial intelligence*. Harper & Row New York, 1979.

Dreyfus, H. Why Heideggerian AI failed and how fixing it would require making it more Heideggerian. *Artificial Intelligence*, 171(18):1137–1160, 2007.

Edelkamp, S. and Reffel, F. Deterministic state space planning with BDDs. Technical report, Computer Science Dept., University of Freiburg, 1999.

Edelkamp, S. Planning with pattern databases. In *European Conference on Planning*, pages 84–90, 2001.

Edelkamp, S. and Helmert, M. Exhibiting knowledge in planning problems to minimize state encoding length. In *European Conference on Planning*, pages 135–147. Springer, 1999.

Edelkamp, S. and Helmert, M. MIPS: The model-checking integrated planning system. *AI magazine*, 22(3):67, 2001.

Edelkamp, S. and Kissmann, P. Optimal symbolic planning with action costs and

preferences. In *International Joint Conference on Artificial Intelligence*, pages 1690–1695, 2009.

Edelkamp, S. and Schroedl, S. *Heuristic search: theory and applications*. Elsevier, 2011.

Enderton, H. B. *A mathematical introduction to logic*. Harcourt Academic press, 2001.

Erol, K., Nau, D., and Subrahmanian, V. Complexity, decidability and undecidability results for domain-independent planning: A detailed analysis. Technical Report CS-TR-2797, UMIACS-TR-91-154, SRC-TR-91-96, University of Maryland, College Park, MD, 1991.

Erol, K., Nau, D., and Subrahmanian, V. When is planning decidable. In *International Conference on AI Planning Systems*, pages 222–227, 1992.

Erol, K., Nau, D. S., and Subrahmanian, V. S. Complexity, decidability and undecidability results for domain-independent planning. *Artificial intelligence*, 76(1): 75–88, 1995.

Ferrer-Mestres, J., Francès, G., and Geffner, H. Planning with state constraints and its application to combined task and motion planning. In *ICAPS Workshop on Planning and Robotics (PlanRob)*, pages 13–22, 2015.

Feydy, T., Somogyi, Z., and Stuckey, P. Half reification and flattening. In *Principles and Practice of Constraint Programming*, pages 286–301. Springer, 2011.

Fikes, R., Hart, P., and Nilsson, N. Learning and executing generalized robot plans. *Artificial intelligence*, 3:251–288, 1972.

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

Fikes, R. E. and Nilsson, N. J. STRIPS, a retrospective. *Artificial Intelligence*, 59 (1):227–232, 1993.

Fox, M. and Long, D. The detection and exploitation of symmetry in planning problems. In *International Joint Conference on Artificial Intelligence*, volume 99, pages 956–961, 1999.

Fox, M. and Long, D. PDDL+: Modeling continuous time dependent effects. In *International NASA Workshop on Planning and Scheduling for Space*, page 34, 2002.

Fox, M. and Long, D. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

Fox, M. and Long, D. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235–297, 2006.

Francès, G. and Geffner, H. Modeling and computation in planning: Better heuristics from more expressive languages. In *International Conference on Automated Planning and Scheduling*, pages 70–78. AAAI Press, 2015.

Francès, G. and Geffner, H. Effective planning with more expressive languages. In *International Joint Conference on Artificial Intelligence*, pages 4155–4159, 2016a.

Francès, G. and Geffner, H. E-STRIPS: Existential quantification in planning and constraint satisfaction. In *International Joint Conference on Artificial Intelligence*, pages 3082–3088, 2016b.

Francès, G., Ramírez, M., Lipovetzky, N., and Geffner, H. Purely declarative action

representations are overrated: Classical planning with simulators. In *International Joint Conference on Artificial Intelligence*, pages 4294–4301, 2017.

Freuder, E. C. Modeling: the final frontier. In *International Conference on The Practical Application of Constraint Technologies and Logic Programming*, pages 15–21, 1999.

Frisch, A. M., Grum, M., Jefferson, C., Hernández, B. M., and Miguel, I. The design of ESSENCE: A constraint language for specifying combinatorial problems. In *International Joint Conference on Artificial Intelligence*, volume 7, pages 80–87, 2007.

Gazen, B. and Knoblock, C. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In *European Conference on Planning*, pages 221–233, 1997.

Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. Answer set solving in practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(3): 1–238, 2012.

Gecode Team. Gecode: Generic constraint development environment, 2006. http://www.gecode.org.

Gefen, A. and Brafman, R. I. The minimal seed set problem. In *International Conference on Automated Planning and Scheduling*, pages 319–322. AAAI Press, 2011.

Geffner, H. Functional STRIPS. In Minker, J., editor, *Logic-Based Artificial Intelligence*, pages 187–205. Kluwer, 2000.

Geffner, H. Perspectives on artificial intelligence planning. In *AAAI Conference on Artificial Intelligence / Innovative Applications of Artificial Intelligence Conference*, pages 1013–1023, 2002.

Geffner, H. PDDL 2.1: Representation vs. computation. *Journal of Artificial Intelligence Research*, 20:139–144, 2003.

Geffner, H. The model-based approach to autonomous behavior: A personal view. In *AAAI Conference on Artificial Intelligence*, pages 1709–1712, 2010.

Geffner, H. and Bonet, B. *A concise introduction to models and methods for automated planning*. Morgan & Claypool, 2013.

Geffner, T. and Geffner, H. Width-based planning for general video-game playing. In *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 23–29, 2015.

Gelfond, M. and Lifschitz, V. Representing action and change by logic programs. *The Journal of Logic Programming*, 17(2-4):301–321, 1993.

Gelfond, M. and Lifschitz, V. Action languages. *Linkoping Electronic Articles in Computer and Information Science*, 3(16), 1998.

Gerevini, A. and Long, D. Plan constraints and preferences in PDDL3: The language of the fifth international planning competition. Technical report, Department of Electronics for Automation, University of Brescia, Italy, 2005.

Gerevini, A. and Serina, I. LPG: A planner based on local search for planning graphs with action costs. In *International Conference on Artificial Intelligence Planning Systems*, volume 2, pages 281–290, 2002.

Gerevini, A. E., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. Deterministic planning in the fifth international planning competition: PDDL3 and experimental

evaluation of the planners. *Artificial Intelligence*, 173(5):619–668, 2009.

Gervet, C. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

Ghallab, M., Nau, D., and Traverso, P. *Automated Planning: theory and practice*. Morgan Kaufmann, 2004.

Giménez, O. and Jonsson, A. The complexity of planning problems with simple causal graphs. *Journal of Artificial Intelligence Research*, 31:319–351, 2008.

Gleitman, L. and Papafragou, A. Language and thought. In Holyoak, K. and Morrison, R., editors, *Cambridge handbook of thinking and reasoning*, pages 633–661. Cambridge University Press, 2005.

Gomes, C. P., Kautz, H., Sabharwal, A., and Selman, B. Satisfiability solvers. In *Handbook of Knowledge Representation*, pages 89–134. Elsevier, 2008.

Gottlob, G., Leone, N., and Scarcello, F. On tractable queries and constraints. In *Database and Expert Systems Applications*, pages 1–15. Springer, 1999.

Green, C. Application of theorem proving to problem solving. In *International joint conference on Artificial intelligence*, pages 219–239. Morgan Kaufmann Publishers Inc., 1969.

Gregory, P., Long, D., Fox, M., and Beck, J. C. Planning modulo theories: extending the planning paradigm. In *International Conference on International Conference on Automated Planning and Scheduling*, pages 65–73. AAAI Press, 2012.

Gupta, N. and Nau, D. S. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2-3):223–254, 1992.

Hart, P., Nilsson, N., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Haslum, P. and Geffner, H. Admissible heuristics for optimal planning. In *Fifth International Conference on AI Planning Systems*, pages 70–82, 2000.

Haslum, P. Reducing accidental complexity in planning problems. In *International Joint Conference on Artificial Intelligence*, pages 1898–1903, 2007.

Haslum, P. Additive and reversed relaxed reachability heuristics revisited. In *The 2008 International Planning Competition Booklet*, 2008.

Haslum, P. Computing genome edit distances using domain-independent planning. In *ICAPS Workshop on Scheduling and Planning Applications*, 2011.

Haslum, P. and Scholz, U. Domain knowledge in planning: Representation and use. In *ICAPS Workshop on PDDL*, 2003.

Haslum, P., Bonet, B., and Geffner, H. New admissible heuristics for domain-independent planning. In *AAAI Conference on Artificial Intelligence*, volume 5, pages 9–13, 2005.

Haslum, P., Botea, A., Helmert, M., Bonet, B., and Koenig, S. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI Conference on Artificial Intelligence*, volume 7, pages 1007–1012, 2007.

Helmert, M. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006a.

Helmert, M., Do, M., and Refanidis, I. The sixth international planning competition, deterministic track, 2008. http://ipc.informatik.uni-freiburg.de, accessed 31 July

2017.

Helmert, M. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262, 2003.

Helmert, M. A planning heuristic based on causal graph analysis. In *International Conference on Automated Planning and Scheduling*, volume 4, pages 161–170. AAAI Press, 2004.

Helmert, M. New complexity results for classical planning benchmarks. In *International Conference on Automated Planning and Scheduling*, pages 52–62. AAAI Press, 2006b.

Helmert, M. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.

Helmert, M. and Domshlak, C. Landmarks, critical paths and abstractions: What's the difference anyway? In *International Conference on Automated Planning and Scheduling*, pages 162–169. AAAI Press, 2009.

Helmert, M. and Mattmüller, R. Accuracy of admissible heuristic functions in selected planning domains. In *AAAI Conference on Artificial Intelligence*, pages 938–943, 2008.

Helmert, M. and Röger, G. How good is almost perfect? In *AAAI Conference on Artificial Intelligence*, volume 8, pages 944–949, 2008.

Helmert, M., Mattmüller, R., and Röger, G. Approximation properties of planning benchmarks. In *European Conference on Artificial Intelligence*, pages 585–589, 2006.

Helmert, M., Haslum, P., and Hoffmann, J. Flexible abstraction heuristics for optimal sequential planning. In *International Conference on Automated Planning and Scheduling*, pages 176–183. AAAI Press, 2007.

Helmert, M., Röger, G., and Karpas, E. Fast downward stone soup: A baseline for building planner portfolios. In *ICAPS Workshop on Planning and Learning*, pages 28–35, 2011.

Hendler, J. A., Tate, A., and Drummond, M. AI planning: Systems and techniques. *AI magazine*, 11(2):61, 1990.

Hertle, A., Dornhege, C., Keller, T., and Nebel, B. Planning with semantic attachments: An object-oriented view. In *European Conference on Artificial Intelligence*, pages 402–407. IOS Press, 2012.

Heusner, M., Keller, T., and Helmert, M. Understanding the search behaviour of greedy best-first search. In *Annual Symposium on Combinatorial Search*, pages 47–55, 2017.

Hoffmann, J. and Nebel, B. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001a.

Hoffmann, J., Porteous, J., and Sebastia, L. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

Hoffmann, J. Local search topology in planning benchmarks: An empirical analysis. In *International Joint Conference on Artificial Intelligence*, pages 453–458, 2001.

Hoffmann, J. Local search topology in planning benchmarks: A theoretical analysis. In *International Conference on Artificial Intelligence Planning Systems*, pages 92–100, 2002.

Hoffmann, J. The metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.

Hoffmann, J. Where "ignoring delete lists" works: local search topology in planning benchmarks. *Journal of Artificial Intelligence Research*, 24:685–758, 2005.

Hoffmann, J. Simulated penetration testing: From "Dijkstra" to "Turing test++". In *International Conference on Automated Planning and Scheduling*, pages 364–372. AAAI Press, 2015.

Hoffmann, J. and Edelkamp, S. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research*, 24:519–579, 2005.

Hoffmann, J. and Nebel, B. What makes the difference between HSP and FF? In *IJCAI Workshop on Empirical Methods in Artificial Intelligence*, 2001b.

Hölldobler, S., Karabaev, E., and Skvortsova, O. FluCaP: a heuristic search planner for first-order MDPs. *Journal of Artificial Intelligence Research*, 27:419–439, 2006.

Holte, R. C., Arneson, B., and Burch, N. PSVN manual (june 20, 2014). Technical Report TR14-03, Computing Science Department, University of Alberta, June 2014.

Hooker, J. N. Testing heuristics: We have it all wrong. *Journal of heuristics*, 1(1): 33–42, 1995.

Hu, Y. and De Giacomo, G. Generalized planning: synthesizing plans that work for multiple environments. In *International Joint Conference on Artificial Intelligence*, pages 918–923. AAAI Press, 2011.

Huth, M. and Ryan, M. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

Imai, T. and Kishimoto, A. A novel technique for avoiding plateaus of greedy best-first search in satisficing planning. In *Annual Symposium on Combinatorial Search*, pages 985–991, 2011.

Ivankovic, F. and Haslum, P. Optimal planning with axioms. In *International Joint Conference on Artificial Intelligence*, pages 1580–1586, 2015.

Ivankovic, F., Haslum, P., Thiébaux, S., Shivashankar, V., and Nau, D. S. Optimal planning with global numerical state constraints. In *International Conference on Automated Planning and Scheduling*, pages 145–153. AAAI Press, 2014.

Jinnai, Y. and Fukunaga, A. Learning to prune dominated action sequences in online black-box planning. In *AAAI Conference on Artificial Intelligence*, pages 839–845, 2017.

Johnson, M., Hofmann, K., Hutton, T., and Bignell, D. The Malmo platform for AI experimentation. In *International Joint Conference on Artificial Intelligence*, pages 4246–4247, 2016.

Jonsson, P. and Bäckström, C. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100(1-2):125–176, 1998.

Junghanns, A. and Schaeffer, J. Domain-dependent single-agent search enhancements. In *International Joint Conference on Artificial Intelligence*, pages 570–577, 1999.

Junghanns, A. and Schaeffer, J. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251, 2001.

Karp, R. M. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

Katz, M. and Domshlak, C. New islands of tractability of cost-optimal planning. *Journal of Artificial Intelligence Research*, 32:203–288, 2008.

Katz, M. and Hoffmann, J. Mercury planner: Pushing the limits of partial delete relaxation. In *The 2014 International Planning Competition Booklet*, 2014.

Katz, M., Hoffmann, J., and Domshlak, C. Who said we need to relax all variables? In *International Conference on Automated Planning and Scheduling*, pages 126–134. AAAI Press, 2013.

Katz, M., Lipovetzky, N., Moshkovich, D., and Tuisov, A. Adapting novelty to classical planning as heuristic search. In *International Conference on Automated Planning and Scheduling*, pages 172–180. AAAI Press, 2017.

Kautz, H. Deconstructing planning as satisfiability. In *National Conference on Artificial Intelligence*, volume 2, page 1524. AAAI Press / MIT Press, 2006.

Kautz, H. and Selman, B. Planning as satisfiability. In *European Conference on Artificial Intelligence*, pages 359–363, 1992.

Kautz, H. and Selman, B. Pushing the envelope: Planning, propositional logic, and stochastic search. In *National Conference on Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference (AAAI, IAAI)*, pages 1194–1201. AAAI Press, 1996.

Kautz, H. and Walser, J. P. State-space planning by integer optimization. In *AAAI Conference on Artificial Intelligence / Innovative Applications of Artificial Intelligence Conference*, pages 526–533, 1999.

Kautz, H., McAllester, D., and Selman, B. Encoding plans in propositional logic. *International Conference on Principles of Knowledge Representation and Reasoning*, 96:374–384, 1996.

Keyder, E. and Geffner, H. Heuristics for planning with action costs revisited. In *European Conference on Artificial Intelligence*, pages 588–592, 2008.

Keyder, E., Hoffmann, J., and Haslum, P. Semi-relaxed plan heuristics. In *International Conference on Automated Planning and Scheduling*, pages 128–136. AAAI Press, 2012.

Koller, A. and Hoffmann, J. Waking up a sleeping rabbit: On natural-language sentence generation with FF. In *International Conference on Automated Planning and Scheduling*, pages 238–241. AAAI Press, 2010.

Korf, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

Korf, R. E. Artificial intelligence search algorithms. In Atallah, M., editor, *Handbook of Algorithms and Theory of Computation*. CRC, 1998.

Kronegger, M., Pfandler, A., and Pichler, R. Parameterized complexity of optimal planning: A detailed map. In *International Joint Conference on Artificial Intelligence*, pages 954–961, 2013.

Lelis, L. H., Zilles, S., and Holte, R. C. Stratified tree search: a novel suboptimal heuristic search algorithm. In *International conference on Autonomous agents and multi-agent systems*, pages 555–562, 2013.

Levesque, H. and Brachman, R. A fundamental tradeoff in knowledge representation and reasoning. In Levesque, H. and Brachman, R., editors, *Readings in Knowledge*

Representation, pages 41–70. Morgan Kaufmann, 1985.

Levesque, H. and Brachman, R. Expressiveness and tractability in knowledge representation and reasoning. *Computational intelligence*, 3(1):78–93, 1987.

Levesque, H., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. B. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1-3):59–83, 1997.

Levesque, H., Pirri, F., and Reiter, R. Foundations for the situation calculus. *Linkoping Electronic Articles in Computer and Information Science*, 3(18), 1998.

Levin, L. A. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

Lifschitz, V. On the semantics of STRIPS. In Georgeff, M. and Lansky, A., editors, *Reasoning about actions and plans*, pages 1–9. Cambridge University Press, 1987.

Lin, F. and Reiter, R. State constraints revisited. *Journal of logic and computation*, 4(5):655–678, 1994.

Lin, F. and Wang, Y. Answer set programming with functions. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 454–464. AAAI Press, 2008.

Lipovetzky, N. *Structure and Inference in Classical Planning*. AI Access, 2014.

Lipovetzky, N. and Geffner, H. Searching for plans with carefully designed probes. In *International Conference on Automated Planning and Scheduling*, pages 154–161. AAAI Press, 2011.

Lipovetzky, N. and Geffner, H. Width and serialization of classical planning problems. In *European Conference on Artificial Intelligence*, pages 540–545. IOS Press, 2012.

Lipovetzky, N. and Geffner, H. Width-based algorithms for classical planning: New results. In *European Conference on Artificial Intelligence*, pages 1059–1060, 2014.

Lipovetzky, N. and Geffner, H. Best-first width search: Exploration and exploitation in classical planning. In *AAAI Conference on Artificial Intelligence*, pages 3590–3596, 2017a.

Lipovetzky, N. and Geffner, H. A polynomial planning algorithm that beats LAMA and FF. In *International Conference on Automated Planning and Scheduling*, pages 195–199. AAAI Press, 2017b.

Lipovetzky, N., Ramirez, M., and Geffner, H. Classical planning with simulators: Results on the Atari video games. In *International Joint Conference on Artificial Intelligence*, pages 1610–1616, 2015.

Long, D. and Fox, M. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

Long, D., Kautz, H., Selman, B., Bonet, B., Geffner, H., Koehler, J., Brenner, M., Hoffmann, J., Rittinger, F., Anderson, C. R., Weld, D. S., Smith, D. E., and Fox, M. The AIPS-98 planning competition. *AI magazine*, 21(2):13–33, 2000.

Mackworth, A. K. Consistency in networks of relations. *Artificial intelligence*, 8(1): 99–118, 1977.

Mansouri, M. and Pecora, F. More knowledge on the table: planning with space, time and resources for robots. In *International Conference on Robotics and Automation*, pages 647–654. IEEE, 2014.

Marques-Silva, J., Lynce, I., and Malik, S. Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, pages 131–153. IOS Press, 2009.

Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P. J., De La Banda, M. G., and Wallace, M. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, 2008.

Martín, M. and Geffner, H. Learning generalized policies from planning examples using concept languages. *Applied Intelligence*, 20(1):9–19, 2004.

Masoumi, A., Soutchanski, M., and Marrella, A. Organic Synthesis as Artificial Intelligence Planning. In *International Semantic Web Applications and Tools for Life Sciences Workshop*, pages 1–15, 2013.

Matloob, R. and Soutchanski, M. Exploring Organic Synthesis with State-of-the-Art Planning Techniques. In *ICAPS Workshop on Scheduling and Planning Applications*, pages 52–61, 2016.

McAllester, D. and Rosenblitt, D. Systematic nonlinear planning. In *AAAI Conference on Artificial Intelligence*, pages 634–639. AAAI Press, 1991.

McCarthy, J. *Programs with common sense*. RLE and MIT Computation Center, 1960.

McCarthy, J. Situations, actions, and causal laws. Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410–417.

McCarthy, J. Generality in artificial intelligence. *Communications of the ACM*, 30 (12):1030–1035, 1987.

McCarthy, J. Elaboration tolerance, 1998. URL `http://www-formal.stanford.edu/jmc/elaboration.pdf`. Unpublished Manuscript.

McCarthy, J. and Hayes, P. J. Some philosophical problems from the standpoint of the artificial intelligence. *Machine Intelligence*, 4:463–502, 1969. Reprinted in Readings in Artificial Intelligence (B.L. Webber and N. J. Nilsson ed.), Tioga, Palo Alto, 1981, pp. 26–46.

McCarthy, J., Minsky, M. L., Rochester, N., and Shannon, C. E. A proposal for the Dartmouth Summer research project on artificial intelligence, August 31, 1955. *AI magazine*, 27(4):12, 2006.

McDermott, D. Regression planning. *International Journal of Intelligent Systems*, 6(4):357–416, 1991.

McDermott, D. A heuristic estimator for means-ends analysis in planning. In *International Conference on Artificial Intelligence Planning Systems*, pages 142–149, 1996.

McDermott, D. Using regression-match graphs to control search in planning. *Artificial Intelligence*, 109(1-2):111–159, 1999.

McDermott, D. The 1998 AI planning systems competition. *AI magazine*, 21(2):35, 2000.

McDermott, D. PDDL 2.1 — the art of the possible? commentary on Fox and Long. *Journal of Artificial Intelligence Research*, 20:145–148, 2003a.

McDermott, D. The formal semantics of processes in PDDL. In *ICAPS Workshop on PDDL*, pages 87–94, 2003b.

McDermott, D. and Hendler, J. Planning: What it is, what it could be, an intro-

duction to the special issue on planning and scheduling. *Artificial Intelligence*, 76 (1):1–16, 1995.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. PDDL – The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, CT, 1998.

McGeoch, C. C. Toward an experimental method for algorithm simulation. *INFORMS Journal on Computing*, 8(1):1–15, 1996.

Minker, J. Introduction to logic-based artificial intelligence. In Minker, J., editor, *Logic-based artificial intelligence*, pages 3–33. Kluwer, 2000.

Miura, S. and Fukunaga, A. Automatically extracting axioms in classical planning. In *AAAI Conference on Artificial Intelligence*, pages 4973–4974, 2017.

Montanari, U. Networks of constraints: Fundamental properties and applications to picture processing. *Information sciences*, 7:95–132, 1974.

Nareyek, A., Freuder, E. C., Fourer, R., Giunchiglia, E., Goldman, R. P., Kautz, H., Rintanen, J., and Tate, A. Constraints and AI planning. *IEEE Intelligent Systems*, 20(2):62–72, 2005.

Nebel, B. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research*, 12:271–315, 2000.

Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., and Tack, G. Minizinc: Towards a standard CP modelling language. *Principles and Practice of Constraint Programming*, pages 529–543, 2007.

Newell, A. and Simon, H. GPS: a program that simulates human thought. In Feigenbaum, E. and Feldman, J., editors, *Computers and Thought*, pages 279–293. McGraw Hill, 1963.

Newell, A. and Simon, H. A. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

Newell, A., Shaw, J. C., and Simon, H. A. Report on a general problem solving program. In *International Conference on Information Processing*, page 64, 1959.

Nilsson, N. *Principles of Artificial Intelligence*. Tioga Press, 1980.

Nilsson, N. The physical symbol system hypothesis: status and prospects. In *50 years of artificial intelligence*, pages 9–17. Springer, 2007.

Novick, L. R. and Bassok, M. Problem solving. In Holyoak, K. and Morrison, R., editors, *Cambridge handbook of thinking and reasoning*, pages 321–349. Cambridge University Press, 2005.

Ohrimenko, O., Stuckey, P. J., and Codish, M. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

OpenAI. Universe software platform. https://universe.openai.com/, 2016.

Papadimitriou, C. H. *Computational complexity*. Addison-Wesley, 1994.

Pearl, J. *Heuristics*. Addison Wesley, 1983.

Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

Pednault, E. P. Formulating multiagent, dynamic world problems in the classical planning framework. In Georgieff, M. P. and Lansky, A. L., editors, *Reasoning About Actions & Plans*, pages 47–82. Cambridge University Press, 1986.

Pednault, E. P. ADL: Exploring the middle ground between STRIPS and the situation calculus. *International Conference on Principles of Knowledge Representation and Reasoning*, 89:324–332, 1989.

Pednault, E. P. ADL and the state-transition model of action. *Journal of logic and computation*, 4(5):467–512, 1994.

Penberthy, J. and Weld, D. UCPOP: A sound, complete, partial order planner for ADL. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 103–114, 1992.

Pennachin, C. and Goertzel, B. Contemporary approaches to artificial general intelligence. In Pennachin, C. and Goertzel, B., editors, *Artificial general intelligence*, chapter 1, pages 1–30. Springer, 2007.

Perez-Liebana, D., Samothrakis, S., Togelius, J., Schaul, T., and Lucas, S. General video game AI: Competition, challenges and opportunities. In *AAAI Conference on Artificial Intelligence*, pages 4335–4337, 2016.

Pohl, I. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1 (3-4):193–204, 1970.

Porco, A., Machado, A., and Bonet, B. Automatic polytime reductions of NP problems into a fragment of STRIPS. In *International Conference on Automated Planning and Scheduling*, pages 178–185. AAAI Press, 2011.

Porco, A., Machado, A., and Bonet, B. Automatic reductions from PH into STRIPS or how to generate short problems with very long solutions. In *International Conference on Automated Planning and Scheduling*, pages 342–346. AAAI Press, 2013.

Porteous, J., Sebastia, L., and Hoffmann, J. On the extraction, ordering, and usage of landmarks in planning. In *European Conference on Planning*, pages 174–182, 2001.

Ramirez, M., Lipovetzky, N., and Muise, C. Lightweight Automated Planning ToolKiT. http://lapkt.org/, 2015.

Ramirez, M., Scala, E., Haslum, P., and Thiebaux, S. Numerical integration and dynamic discretization in heuristic search planning over hybrid domains. *arXiv preprint arXiv:1703.04232*, 2017.

Rardin, R. L. and Uzsoy, R. Experimental evaluation of heuristic optimization algorithms: A tutorial. *Journal of Heuristics*, 7(3):261–304, 2001.

Rautenberg, W. *A concise introduction to mathematical logic*. Springer, 2006.

Régin, J.-C. Global constraints: A survey. In van Hentenryck, P. and Milano, M., editors, *Hybrid Optimization: The Ten Years of CPAIOR*, pages 63–134. Springer, 2011.

Reiter, R. On closed world data bases. In Gallaire, H. and Minker, J., editors, *Logic and Data Bases*, pages 55–76. Springer, 1978.

Reiter, R. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. MIT press, 2001.

Richter, S. and Westphal, M. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39(1):127–177, 2010.

Richter, S. and Helmert, M. Preferred operators and deferred evaluation in satisficing planning. In *International Conference on Automated Planning and Scheduling*,

pages 273–280. AAAI Press, 2009.

Richter, S., Helmert, M., and Westphal, M. Landmarks revisited. In *AAAI Conference on Artificial Intelligence*, volume 8, pages 975–982, 2008.

Riddle, P., Barley, M., Franco, S., and Douglas, J. Analysis of bagged representations in PDDL. In *ICAPS Workshop on Heuristics and Search for Domain-independent Planning*, pages 71–79, 2015a.

Riddle, P., Douglas, J., Barley, M., and Franco, S. Improving performance by reformulating PDDL into a bagged representation. *ICAPS Workshop on Heuristics and Search for Domain-independent Planning*, page 28, 2016.

Riddle, P. J., Holte, R. C., and Barley, M. W. Does representation matter in the planning competition? In *Symposium of Abstraction, Reformulation, and Approximation*, 2011.

Riddle, P. J., Barley, M. W., Franco, S., and Douglas, J. Automated transformation of PDDL representations. In *Annual Symposium on Combinatorial Search*, pages 214–215, 2015b.

Rintanen, J. Impact of modeling languages on the theory and practice in planning research. In *AAAI Conference on Artificial Intelligence*, pages 4052–4056, 2015.

Rintanen, J. and Jungholt, H. Numeric state variables in constraint-based planning. In *European Conference on Planning*, pages 109–121, 1999.

Rintanen, J. Unified definition of heuristics for classical planning. In *European Conference on Artificial Intelligence*, pages 600–604. IOS Press, 2006.

Rintanen, J. Heuristics for planning with SAT and expressive action definitions. In *International Conference on Automated Planning and Scheduling*, pages 210–217. AAAI Press, 2011.

Rintanen, J. Planning as satisfiability: Heuristics. *Artificial Intelligence*, 193:45–86, 2012.

Rintanen, J. Madagascar: Scalable planning with SAT. In *The 2014 International Planning Competition Booklet*, 2014.

Rintanen, J., Heljanko, K., and Niemelä, I. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.

Röger, G. and Helmert, M. The more, the merrier: combining heuristic estimators for satisficing planning. In *International Conference on International Conference on Automated Planning and Scheduling*, pages 246–249. AAAI Press, 2010.

Rossi, F., Van Beek, P., and Walsh, T. *Handbook of constraint programming*. Elsevier, 2006.

Rossi, F., Van Beek, P., and Walsh, T. Constraint programming. In Van Harmelen, F., Lifschitz, V., and Porter, B., editors, *Handbook of Knowledge Representation*, chapter 4, pages 181–211. Elsevier, 2008.

Russell, S. and Norvig, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009. Original work published in 1994.

Sacerdoti, E. D. Planning in a hierarchy of abstraction spaces. *Artificial intelligence*, 5(2):115–135, 1974.

Sacerdoti, E. D. The nonlinear nature of plans. In *International joint conference on Artificial intelligence*, pages 206–214. Morgan Kaufmann, 1975.

Scala, E., Haslum, P., Thiébaux, S., and Ramírez, M. Interval-based relaxation for

general numeric planning. In *European Conference on Artificial Intelligence*, pages 655–663, 2016a.

Scala, E., Ramirez, M., Haslum, P., and Thiebaux, S. Numeric planning with disjunctive global constraints via SMT. In *International Conference on Automated Planning and Scheduling*, pages 276–284. AAAI Press, 2016b.

Schaul, T. A video game description language for model-based or interactive learning. In *IEEE Conference on Computational Intelligence and Games*, pages 1–8, 2013.

Seligman, M., Railton, P., Baumeister, R., and Sripada, C. *Homo prospectus*. Oxford University Press, 2016.

Shannon, C. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

Shleyfman, A., Tuisov, A., and Domshlak, C. Blind search for Atari-like online planning revisited. In *International Joint Conference on Artificial Intelligence*, pages 3251–3257, 2016.

Slaney, J. and Thiébaux, S. Blocks World revisited. *Artificial Intelligence*, 125(1-2): 119–153, 2001.

Smith, B. M. Modelling. *Foundations of Artificial Intelligence*, 2:377–406, 2006.

Smith, D. E. The case for durative actions: A commentary on PDDL 2.1. *Journal of Artificial Intelligence Research*, 20:149–154, 2003.

Soemers, D., Sironi, C., Schuster, T., and Winands, M. Enhancements for real-time Monte-Carlo tree search in general video game playing. In *IEEE Conference on Computational Intelligence and Games*, pages 1–8, 2016.

Son, T. C., Tu, P. H., Gelfond, M., and Morales, A. Conformant planning for domains with constraints: A new approach. In *AAAI Conference on Artificial Intelligence*, pages 1211–1216, 2005.

Stuckey, P. J. Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving. In *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming (CPAIOR)*, volume 10, pages 5–9. Springer, 2010.

Subrahmanian, V. and Zaniolo, C. Relating stable models and AI planning domains. In *International Conference on Logic Programming*, pages 233–247, 1995.

Tate, A. Generating project networks. In *International joint conference on Artificial intelligence*, pages 888–893. Morgan Kaufmann, 1977.

Thiébaux, S., Hoffmann, J., and Nebel, B. In defense of PDDL axioms. *Artificial Intelligence*, 168(1–2):38–69, 2005.

Torralba, Á., Alcázar, V., Kissmann, P., and Edelkamp, S. Efficient symbolic search for cost-optimal planning. *Artificial Intelligence*, 242:52–79, 2017.

Turing, A. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

Vallati, M., Chrpa, L., Grzes, M., McCluskey, T. L., Roberts, M., and Sanner, S. The 2014 international planning competition: Progress and trends. *AI Magazine*, 36(3):90–98, 2015a.

Vallati, M., Hutter, F., Chrpa, L., and McCluskey, T. L. On the effective configuration of planning domain models. In *International Joint Conference on Artificial Intelligence*, pages 1704–1711, 2015b.

Van Hentenryck, P. and Carillon, J.-P. Generality versus specificity: An experience

with AI and OR techniques. In *AAAI Conference on Artificial Intelligence*, pages 660–664, 1988.

van Hoeve, W.-J. and Katriel, I. Global constraints. *Handbook of constraint programming*, pages 169–208, 2006.

van Hoeve, W. The alldifferent constraint: A survey. *Annual Workshop of the ERCIM Working Group on Constraints*, 2001.

Vidal, V. A lookahead strategy for heuristic search planning. In *International Conference on Automated Planning and Scheduling*, pages 150–160. AAAI Press, 2004a.

Vidal, V. The YAHSP planning system: Forward heuristic search with lookahead plans analysis. In *The 2004 International Planning Competition Booklet*, pages 56–58, 2004b.

Vidal, V. YAHSP2: Keep it simple, stupid. In *The 2011 International Planning Competition Booklet*, pages 83–90, 2011.

Vossen, T., Ball, M., Lotem, A., and Nau, D. On the use of integer programming models in AI planning. In *International joint conference on Artifical intelligence*, pages 304–309. Morgan Kaufmann, 1999.

Walsh, T. SAT v CSP. In *Principles and Practice of Constraint Programming*, pages 441–456. Springer, 2000.

Weld, D. and Etzioni, O. The first law of robotics (a call to arms). In *AAAI Conference on Artificial Intelligence*, volume 94, pages 1042–1047, 1994.

Weld, D. S. An introduction to least commitment planning. *AI magazine*, 15(4):27, 1994.

Weld, D. S. Recent advances in AI planning. *AI magazine*, 20(2):93, 1999.

Wilt, C. M. and Ruml, W. Building a heuristic for greedy search. In *Annual Symposium on Combinatorial Search*, pages 131–139, 2015.

Xie, F., Müller, M., and Holte, R. Jasper: the art of exploration in greedy best first search. In *The 2014 International Planning Competition Booklet*, 2014a.

Xie, F., Müller, M., Holte, R., and Imai, T. Type-based exploration with multiple search queues for satisficing planning. In *AAAI Conference on Artificial Intelligence*, pages 2395–2402, 2014b.

Xie, F., Müller, M., and Holte, R. Adding local exploration to greedy best-first search in satisficing planning. In *AAAI Conference on Artificial Intelligence*, pages 2388–2394, 2014c.

Yoon, S. W., Fern, A., and Givan, R. FF-Replan: A baseline for probabilistic planning. In *International Conference on Automated Planning and Scheduling*, volume 7, pages 352–359. AAAI Press, 2007.

Younes, H. L. and Simmons, R. G. On the role of ground actions in refinement planning. In *International Conference on Artificial Intelligence Planning Systems*, pages 54–62, 2002.