Universidad Carlos III de Madrid

TESIS DOCTORAL

Generation and Exploitation of Intermediate Goals in Automated Planning

Autor

Vidal Alcázar Saiz

Directores

Dr. D. Daniel Borrajo Millán y Dra. Dña. Susana Fernández Arregui

Departamento de Informática. Escuela Politécnica Superior

Leganés, 11 de Julio de 2014

TESIS DOCTORAL

GENERATION AND EXPLOITATION OF INTERMEDIATE GOALS IN AUTOMATED PLANNING

Autor: Vidal Alcázar Saiz

Directores: Dr. D. Daniel Borrajo Millán y Dra. Dña. Susana Fernández Arregui

Tribunal Calificador		Firma
Presidente:	....	....
Vocal:	....	....
Secretario:	....	....

Calificación: ....

Leganés, ... de .... de 2014

Contents

List of Figures

Acknowledgements

First and foremost, I'd like to thank my advisors Daniel and Susana, who gave me not only support and guidance throughout the thesis, but also complete freedom to pursue the lines I wanted at my pace. Thanks a lot for your patience and understanding, and for the trust you put in me, I can only hope I'll be working hand in hand with people half as kind as you. Special mention to Carlos too, who was my advisor for a short time and from whom I only received praise and encouragement.

Of course, many many thanks to the guys at the lab too. Álvaro, Sergio, Isa, Jesús, Javi, Nere, Moisés, Eze, Pulido, JC,... thanks a lot for forgiving my sometimes harsh remarks and my pet peeves (you know I only meant good! haha), it was a great experience to have a workplace I was actually looking forward to going to. Also I've lost count of all the hearty laughs I've had with you guys, proud loudest researchers of the building! I hope those beers and those trips together won't stop once we are not part of the group anymore.

I won't forget the rest of the people at the PLG either. Tomás, Sergio, Raquel, Fernando, Héctor, Ángel, Dani, Javi, Neli, Rocío, Dionisis, Leila, Grecia,... you all made the PLG that little gem that stood out among the rest of the groups, a group in which people always came first and in which there was always time and occasion for a friendly get-together. I don't think a PhD. student could wish for a better place to spend the years of his thesis at.

I also want to thank all the people that warmly received me at Pittsburgh and Freiburg. Thanks Manuela and Malte for listening to my banter through so many meetings and for providing me with your opinions and criticisms, each of you in your own particular way. Thanks a lot to the rest of the people of the groups, specially the guys at Freiburg; I don't think I've ever learnt so much about something as during my six months with you, a really humbling and eye-opening experience.

I'd like to reach out to the planning community as a whole too, a small and welcoming community full of incredible people that has taught me a lot. Also I want to include in these acknowledgements all the institutions that provided me with funding for my activities, without their generous contributions I couldn't have devoted myself to this beautiful art that research is.

It goes without saying, but many thanks to my family and friends, I wouldn't be the person I am without you. And sorry too to those that I may have forgotten; you know me all too well, so you'll know I'm writing these lines late at night.

Vidal.

Resumen

En planificación automática, los planificadores independientes de dominio a menudo escalan pobremente. Esto se debe a la explosión exponencial del esfuerzo necesario para resolver una tarea de planificación según su tamaño incrementa. Uno de las formas más populares de abordar este problema es dividiendo el problema de planificación en varios problemas más pequeños.

Para separar la tarea en tareas más pequeñas, hay que encontrar métodos independientes de dominio capaces de derivar metas intermedias. En esta tesis se estudiarán diferentes aproximaciones que generen y aprovechen metas intermedias, sin limitarnos a una mera subdivisión del problema original. Tres líneas de investigación serán exploradas. La primera trata sobre regresión, primero encarando sus limitaciones y después usándola tanto en búsqueda bidireccional como en nuevas heurísticas basadas en metas intermedias. En la segunda línea proponemos muestrear aleatoriamente el espacio de búsqueda y usar las submetas muestreadas aleatoriamente en un algoritmo basado en árboles aleatorios que balancea exploración y explotación de forma efectiva. Finalmente, en la tercera línea de investigación estudiamos las propiedades del grafo de landmarks, el cual representa las restricciones de precedencia entre submetas de la tarea. Como contribución, proponemos diferentes caracterizaciones del grafo de landmarks que mejoran su formulación original proporcionando más información, tanto propiedades formales de la tarea como ordenaciones de submetas más informadas aprovechables por planificadores que emplean landmarks.

Abstract

In automated planning, domain-independent planners often scale poorly. This is due to the exponential blow up of the effort necessary to solve a planning task as its size increases. One of the most popular ways of addressing this problem is splitting the planning problem into several smaller ones. Each subproblem is in theory exponentially easier to solve than the original one, so planners that divide the original task will tend to scale much better.

To divide the task into smaller ones, we need to find domain-independent methods to derive intermediate goals. In this thesis we will study different approaches that generate and exploit intermediate goals, without limiting ourselves to simply splitting the original problem. Three main lines of research will be pursued. The first one deals with regression, first tackling its shortcomings and then using it both in bidirectional search and as a way to derive novel heuristics based on intermediate goals. In the second one we propose sampling the search space randomly and using the randomly-sampled subgoals in a tree-like algorithms that effectively balances exploration and exploitation. Finally, in the third one we study the properties of the landmark graph, which represents precedence constraints among subgoals of the task. As a contribution, we propose different characterizations of the landmark graph that improve over its original formulation by providing more information, both formal properties of the task and finer orderings of subgoals exploitable by planners that already use landmarks.

Chapter 1

Introduction

One of the main characteristics that defines humans as opposed to individuals of other species is the capacity of abstraction and deliberation. This skill, intrinsic to the human species, allows us to reason about the outcome of our actions and the means that may be necessary to achieve a given goal. Modern examples of such problems would be planning a weekend out or cooking a certain dish for which some ingredients must be bought. Being such an important trait, it is only natural that researchers in several fields would eventually turn their attention to the understanding and replicating of this behavior, which ultimately led to the creation of the area of Automated Planning (AP).

AP can be traced back to the late '50s, when electronic general-purpose computers were still at their early stages of development. It was initially conceived as a general area that included theorem proving and search among other computation paradigms, thus representing a substantial part of what the proponents of the time defined as Artificial Intelligence. In fact, one of the first planning systems ever designed, the General Problem Solver (Newell et al., 1959), was intended to solve any kind of problem that could be formally defined, which gives an idea of the ambitious aim of the early conception of AP. Its scope was greatly reduced afterwards, but AP still stands as one of the trademarks of the definition of Artificial Intelligence.

Currently, AP can be defined as the task of finding a sequence of actions (commonly called plan) that achieves a particular set of goals from a given initial state. Early on, problems in AP were often modeled as theorem proving problems, consistent with the background of the researchers that worked in AP at that time (Mccarthy and Hayes, 1969). During the '70s STRIPS (STanfoRd Institute Problem Solver) (Fikes and Nilsson, 1971) was developed as the deliberative component of the software that controlled Shakey, an autonomous robot designed to accomplish a broad range of tasks. As opposed to prior models, STRIPS received as an input a planning task defined in terms of predicates, operators, initial state and goals. This model quickly became very popular in the field due to its simplicity and how well it represented the planning paradigm. In fact, this is so to the point that STRIPS is commonly accepted as almost equivalent to classical planning, the simplest version of AP, and is still very relevant in current research in the area.

The embrace of STRIPS by the community marked the beginning of modern AP. From then on, the field developed in two ways: expressiveness and efficiency. Regarding expressiveness, the need for more expressive representations originated the development of both the Action Definition Language (ADL) (Pednault, 1989) and the Planning Domain Definition Language (PDDL) (Fox and Long, 2003; McDermott et al., 1998), built upon STRIPS and which allowed the representation of conditional effects, quantification, numerical values, time, preferences, probabilities and more throughout their versions. Currently few planners support all features of PDDL, so most of the fundamental research in the area is done in classical planning; that is, a propositional representation of the problem with a few other features like object types, action costs and conditional effects.

From the point of view of its theoretical complexity, AP is PSPACE-complete for both the optimal and suboptimal case (Bylander, 1994). This means that in the worst case the difficulty of the task increases exponentially as the task becomes bigger. Thus, planning can be very hard, but it tells little about the performance of planners on average. The most crucial point regarding AP is that it is normally considered a domain-independent process: the planner has no prior knowledge about the task to solve and no other input is given to the system by humans. Therefore, some problems which are easier than PSPACE-complete are actually hard for current planners to solve when modeled in PDDL. Furthermore, the difference between solving a problem with an *ad-hoc* solver and with a planner is usually very big, with *ad-hoc* solvers beating domain-independent planners by a great margin in most cases. For instance, Blocksworld, a domain trivial for humans and *ad-hoc* solvers alike and that is often used as a benchmark in AP, remains a hard problem for planners despite its long history in the field. In fact, planners were not able to solve problems with more that 10-15 blocks less than 20 years ago and currently it is still a challenging domain in terms of how state-of-the-art planners scale as the number of blocks increases.

Over the years, a broad number of approaches have been employed in AP: partial-order planning (Penberthy and Weld, 1992; Younes and Simmons, 2003), search in planning graphs (Blum and Furst, 1997), means-ends analysis (Veloso et al., 1995), reachability heuristics (Bonet and Geffner, 2001), stochastic local search (Gerevini and Serina, 2002), dynamic programing with SAT encodings of the problems (Kautz et al., 2006),... A very promising approach is dividing the problem in several sequential subproblems, also known as factored planning (Brafman, 2006). A similar approach is the use of techniques that exploit information regarding facts that must be true at some point in the plan, usually in the form of landmarks (Hoffmann et al., 2004). Factored planning is a very appealing technique since it reduces the depth of the search space, which may translate into an exponential gain in performance; landmarks and other similar techniques have already proved to be very useful, as the International Planning Competitions have shown[1]. Nevertheless, depending on the domain these kind of approaches do not always improve the efficiency of planners, as finding meaningful subgoals to partition the problem or from which to derive heuristics may not be easy.

Due to the potential that finding critical subgoals can offer, in this thesis we aim to further develop this line of research. For this we study how to generate and exploit intermediate goals in a domain-independent way, although we do not limit ourselves to subgoals. Throughout the thesis we analyze the shortcomings of several techniques related with intermediate goals, and propose the use of algorithms tangentially related to intermediate goals, like bidirectional search.

The main contributions are:

- analyzing alternative methods to find relevant subgoals, like regression, and push the state of the art regarding these methods
- using alternative algorithms that randomly sample the space and work with subproblems of the real problem
- further exploiting already known information in the landmark graph to enhance the search

These three lines of research are related in that all three have as the core of their research domain-independent intermediate goals, and that they all employ several properties of the planning task, like state invariants, to ensure the practicality of the presented approaches.

This document is organized as follows: In Part I, we introduce the state of the art, with a special emphasis on the works that constitute the basis of the thesis; in Part II we state the major objectives of the thesis; in Part III we study how regression may help the planning process both as backward search with modern techniques and as a way of generating intermediate goals; in Part IV we analyze the challenges of sampling implicit search spaces and propose ways of adapting random sampling trees to automated planning; in Part V we study the caveats of the landmark graph in its current form and propose two different characterizations (compiling it into a SAT instance and formalizing it as an abstraction of the original problem), along with different ways of exploiting the additional information they offer.

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

Part I

State of the Art

Chapter 2

Background

This chapter is a review of background related to the main topics addressed in this thesis. After a small introduction AP will be given and the formalizations employed in the thesis described. Next, we present the most common modeling languages used in AP, make an overview of the history of AP and analyze a set of techniques used throughout different chapters of the thesis.

2.1 Introduction

General solvers are systems designed to find a solution to any problem described in a high level modeling language. The necessary components of such a system are:

- Conceptual Model: the common formulation of all the relevant problems
- Representative Language: the language used to model the problems belonging to the conceptual model
- Algorithm: the set of techniques used by the solver to find a solution

AP is a prime example of general problem solving. In particular, all the problems in AP consist on choosing actions to create a plan such that a given set of goals can be achieved by executing the actions of the plan from the initial state. In most cases, some general assumptions are made for the sake of simplicity. The following are the common assumptions made in the field in regards to common planning tasks:

- Finite world: the world has a finite set of states
- Deterministic: applying an action in a given state leads to a single other state in all cases
- Static: no external changes occur in the domain, all the changes come from the execution of actions

- Fully observable: all the information about the state of the world is known (or can be inferred) by the planner at all times

Others models of planning exist: temporal planning allows actions with different durations, fluents encode numeric facts, probabilistic planning allows non-deterministic actions, contingent planning implies partial observability, conformant planning requires finding a safe plan in a non-observable environment, etc. In this proposal, though, these models will not be taken into account.

The quality of a solution is inversely proportional to the cost of the plan. The cost of a plan is the sum of the cost of the actions it is composed of. Although the cost of an action may depend on the state it is executed at, we will assume that the cost of the actions is fixed.

Taking into account this measure of quality, another possible distinction is between optimal planning and satisficing (suboptimal) planning. In the former, an optimal plan (a least-cost plan) must be found, whereas in the latter any solution plan is valid, although those with lower cost are preferred.

2.2 Formalization of AP

There exist several ways of formalizing a planning task. Although planning problem are usually encoded as transition systems, the formalization depends on the underlying techniques used to solve or derive properties of the task. In general planning problems can be formalized as either search problems in the space of states or search problems in the space of plans. Both types of formalizations have been used by planners; however, state-space search in the form of heuristic search (Bonet and Geffner, 2001) has obtained better results overall. In this thesis we will work exclusively with state-space planning techniques.

Among state-space formalisms, we can make the distinction between propositional and multi-valued formalizations. Both kinds of formalizations will be used throughout the thesis, so we describe them next. However, by default we will employ a propositional formalization and assume that propositions and fluents are equivalent unless otherwise stated.

2.2.1 Propositional Formalization

A propositional formalization of a planning task is defined as a tuple *P=(S,A,I,G)*, where *S* is a set of atomic propositions (also known as *facts*), *A* is the set of grounded actions derived from the operators of the domain (described in the following section), *I* $\subseteq$ *S* is the initial state, *G* $\subseteq$ *S* the set of goal propositions. We also define *c(a)* as the cost of applying action $a \in A$ in any state $s \subseteq S$. Each action $a \in A$ is defined as a triple *(pre(a), add(a), del(a))* (preconditions, add effects and delete effects) where *pre(a), add(a), del(a)* $\subseteq$ *S*.

Finding a solution to a planning problem *P* consists of generating a sequence of actions $(a_1, a_2, \ldots, a_n)$ where $a_i \in A$. The solution plan is related to a sequence of states $(s_0, s_1, s_2, \ldots, s_n)$ of the plan such that $s_i \subseteq S$, $s_0 = I$, $G \subseteq s_n$ and s_i results from executing the action a_i in the state s_{i-1}, $\forall i = 1..n$. The cost of a plan is defined by $\sum_{i=1}^{n} c(a_i)$.

An example is shown in Figure 2.1, which depicts Sussman's anomaly, a well-known task of the *Blocksworld* domain. This planning task consists in stacking blocks on top of each other using a robotic arm. The initial state is the one shown on the left and the goals are stacking A on top of B and B on top of C. S is formed by propositions of the type *(on X Y)*, *(on-table X)*, *(clear X)*, *(holding X)* and *(arm-empty)*. In this problem, G = *(on A B)* $\wedge$ *(on B C)* and I = *(on C A)* $\wedge$ *(clear C)* $\wedge$ *(on-table A)* $\wedge$ *(on-table B)* $\wedge$ *(clear B)* $\wedge$ *(arm-empty)*. The actions of A are of the form *(stack X Y)*, *(unstack X Y)*, *(pick-up X)* and *(put-down X)*. The optimal plan is *(unstack C A)*, *(put-down C)*, *(pick-up B)*, *(stack B C)*, *(pick-up A)* and *(stack A B)*.

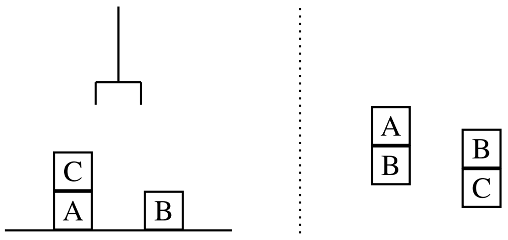


Figure 2.1: Deterministic planning problem of the domain *Blocksworld* in which blocks must be stacked using a robotic arm.

2.2.2 Multi-Valued Formalizations

The two most common multi-valued formalizations are SAS$^+$ (Bäckström and Nebel, 1995) and the Finite-Domain Representation proposed by Helmert (2006), based on SAS$^+$. In this work we will refer only to SAS$^+$. A planning task in SAS$^+$ is defined as a tuple Π=($\mathcal{V}$,s_0,$s_\star$,$\mathcal{O}$). $\mathcal{V}$ is a set of state variables, and every variable $v \in \mathcal{V}$ has an associated extended domain $D_v^+ = D_v \cup \{\mathbf{u}\}$ composed of the regular domain of each variable, D_v, and the undefined value $\mathbf{u}$ (used to denote when the value is unknown). The total state space is defined as $S_{\mathcal{V}}^+ = D_{v_0}^+ \times \ldots \times D_{v_n}^+$ and the value of a variable $v \in \mathcal{V}$ in a given state s, also known as *fluent*, is defined as $s[v]$. Partial states are states in which at least a fluent $s[v_i] = \mathbf{u}$. s_0 is the initial state, defined over $\mathcal{V}$ such that $s_0[v_i] \neq \mathbf{u}\ \forall v_i \in \mathcal{V}$ under full observability of states. $s_\star$ is the (commonly partial) state that defines the goals, defined over $\mathcal{V}$ such that $s_\star[v_i] \in D_{v_n}^+\ \forall v_i \in \mathcal{V}$. $\mathcal{O}$ is a set of operators (actions), where each operator is a tuple $o = (pre(o), post(o), prev(o))$, where $pre(o), post(o), prev(o) \in S_v^+$ represent the *pre-*, *post-* and *prevail-conditions* respectively. The *preconditions* of $o \in \mathcal{O}$ are fluents that must be true prior to the application of o and become not true after its application; the *postconditions* of o are fluents that are not true prior to the application of o and become true after its application; the *prevail conditions* of o are fluents that must be true before and after the application of o. Therefore, an action $o \in \mathcal{O}$ is applicable in a state s if $\forall v_i \in \mathcal{V} : (prev(o)[v_i] = \mathbf{u} \vee prev(o)[v_i] = s[v_i]) \wedge (pre(o)[v_i] = \mathbf{u} \vee pre(o)[v_i] = s[v_i])$. The resulting state s' from the application of o in s is equal to s except that $\forall v_i \in \mathcal{V}\ s.t.\ post(o)[v_i] \neq \mathbf{u} : s'[v_i] = post(o)[v_i]$.

Analogous to the propositional formalization, a solution plan $(a_1, a_2, \ldots, a_n)$ where $a_i \in \mathcal{O}$ is related to a sequence of states $(s_0, s_1, s_2, \ldots, s_n)$ such that $s_\star \subseteq s_n$ and s_i results from executing the action a_i in the state s_{i-1}, $\forall i = 1..n$. This sequence of states is often referred to as the solution path.

Fluents in SAS$^+$ are extrapolable to propositions in a propositional representation. For instance, regarding actions, preconditions in SAS$^+$ are preconditions deleted by the action, postconditions are positive effects and prevail conditions are preconditions not deleted by the action. In this work we assume that every concept described in terms of fluents is also relevant to propositions.

To illustrate how multi-valued formalizations work, let's take a look at the example shown in Figure 2.1. In this case we have the following variables: a variable per block that indicates where the block is (upon which block it is placed or whether it is on the table or whether the arm is holding it); a variable per block that indicates whether it is clear or not; and an additional variable that represents whether the arm is empty or not. For instance, for *block A* we have a variable $v_a \in \mathcal{V}$ such that D_{v_a} = {*(on A B), (on A C), (on-table A), (holding A)*} plus a variable $v'_a \in \mathcal{V}$ such that $D_{v'_a}$ = {*(clear A)*, ⟨*none of those*⟩}, in which ⟨*none of those*⟩ corresponds to the negation of *(clear A)*.

To know the details of how multi-valued formalizations are obtained we refer the reader to the insightful paper by Malte Helmert (2006). However, some invariants closely related to multi-valued formalizations will be throughly analyzed in Section 2.5 because of their relevance for this thesis.

2.3 Modeling Languages

A planning modeling language is a notation that allows a syntactic representation of a planning problem. The planning modeling languages are based on variants of first order logic and describe the different features of the planning problem. Depending on its expressiveness, a modeling language will be able to encode complex features like time, real numbers and probabilities.

2.3.1 STRIPS

The STRIPS planner was initially developed by Richard Fikes and Nils Nilsson in 1971 at SRI International for its use by the robot "Shakey". This planner received the planning task as an input modeled in a language based on LISP, which was later called STRIPS as well (Fikes and Nilsson, 1971). This model was particularly apt for describing planning tasks and due to its popularity became the most popular model for subsequent planners and is still the base of most modern planning languages.

A major contribution made by STRIPS is adopting the *Closed World Assumption* (Reiter, 1977) to handle the Frame Problem. The Frame Problem (McCarthy and Hayes, 1969) is the problem of expressing a dynamical domain in logic without explicitly specifying which literals are not affected by the definition of a state or an action. The *Closed World Assumption* means that any formula not explicitly asserted in a state is taken to be false, which avoids the constant specification of all negated atomic formulæ. For example, if I does not contain a proposition $p \in S$, then p is false in the initial state. The *STRIPS Assumption* can be seen as a reformulation of the *Closed World Assumption* regarding actions: it assumes that actions only change a small part of the world and so, if a proposition $p \in S$ is not mentioned in the effects of an action $a \in A$ ($p \notin add(a) \land p \notin del(a)$) and a is applied in state $s \subseteq S$ to obtain $s' \subseteq S$, then p has the same value in s and s'.

The definition of a problem usually requires two inputs: the domain and the problem itself. The domain describes the predicates of the problem (which are usually instantiated to create the set of propositions S) and the operators of the problem (which again are usually instantiated to create the set of actions A). In Table 2.1 we show the operators of the domain *Blocksworld* in STRIPS:

Operator	Preconditions	Deletes	Adds

stack(x,y)	clear(y)
	holding(x)
		clear(y)
		holding(x)
			handempty
			on(x,y)

unstack(x,y)	handempty
	on(x,y)
	clear(x)
		handempty
		on(x,y)
			clear(y)
			holding(x)

pick-up(x)	clear(x)
	handempty
	ontable(x)
		clear(x)
		handempty
		ontable(x)
			holding(x)

put-down(x)	holding(x)	holding(x)	handempty
			ontable(x)
			clear(x)

Table 2.1: STRIPS definition of the *Blocksworld* domain.

The problem describes the objects of the problem, the propositions true in the initial state I and the goal set of propositions G. Here we show the description of the problem shown in Figure 2.1:

```
Objects: A B C
Init: (clear C) (clear B) (on C A)
      (ontable A) (ontable B) (handempty)
Goal: (on A B) (on B C)
```

The reason why domain and problems are split into two different inputs is because often multiple problems may belong to the same domain. In this case different problems may have a different number of blocks, a different initial state and/or a different set of goals, but they all share the predicates and the operators that define the actions. Hence, a domain may be described in terms of predicates and operators once and used for an arbitrary number of problems.

2.3.2 ADL

The Action Description Language (ADL) (Pednault, 1989) is one of the first extension of STRIPS. There are two main contributions of ADL: the preconditions of the actions and the goals can be expressed not just as conjunction of propositions but also as disjunctions or quantified formulæ; and actions can have conditional effects.

2.3.3 PDDL

The Planning Domain Definition Language (PDDL) (McDermott et al., 1998) was created to standardize the way planning tasks are defined. The aim of this language is to allow a fair comparison of planners, because with a common language a proper benchmark usable by all the planners can be easily created. In fact, PDDL was proposed as the language of the first International Planning Competition, whose motivation is to compare state-of-the-art techniques and further encourage the development of more efficient planners. Throughout the years there have been several versions of PDDL, each of which added new language features to the ones supported by previous versions:

- PDDL1.7 (IPC1 and IPC2): it supported STRIPS, ADL and types of objects.
- PDDL2.1 (IPC3): it added numeric variables which could be modified by actions of the problem and also added a new type of action: durative actions. Durative actions have both discrete and continuous effects and allow representing time in the planning task.
- PDDL2.2 (IPC4): it extended PDDL2.1 with derived predicates (also known as axioms), which are propositions that become true when their antecedents are made true. Furthermore, it introduced timed initial literals, propositions that are initially false but that become true after a given amount of time independently of the actions of the plan.
- PDDL3.0 (IPC5): it allowed the use of soft goals encoded as preferences (among other things), which would yield a reward if they were true at the end of the plan.

Despite the broad expressiveness of the latest versions of PDDL, most planners work exclusively with what is commonly known as classical planning, that is, STRIPS plus action costs, object types and the equality operator. Furthermore, several works have proposed compiling the features of PDDL away by transforming the task into a classical planning one, like compiling away soft goals with actions that achieve those soft goals with no reward (Keyder and Geffner, 2009a), which has deterred to some extent the popularization of the newest features of PDDL.

2.3.4 Other Versions of PDDL

Other variations of PDDL have been proposed to deal with the aforementioned planning paradigms that drop some of the key assumptions normally made in planning. In particular, PDDL versions exist for probabilistic, contingent and conformant planning. A prime example of these variations is Probabilistic PDDL (PPDDL) (Younes and Littman, 2004).

2.4 History

Since the inception of AP multiple algorithms have been employed to build domain-independent planners. Most of these algorithms are search algorithms, which try to find a path in an implicit directed graph that reaches a goal node from the initial node. There are two common elements in search that must be defined before applying a search algorithm: the search space and the direction of the search.

As mentioned in Section 2.2, in AP the search space can be either a space of states or a state of plans. In state spaces every node of the graph corresponds to a state that represents the state in the world. Arcs represent transitions between states, each of which is originated from an action of the model. In plan spaces, though, the nodes are partially specified plans and the arcs correspond to plan refinement operations, which add or remove actions in the parent state.

In terms of direction of the search, three possibilities exist: the search can be performed forward (also called progression), backward (also called regression) and in both directions (also called bidirectional search). Each approach has distinct advantages and disadvantages, part of which will be studied in the thesis.

Depending on the search space and the direction of the search, different search algorithms and heuristics have shown to be more efficient than others. We include here in chronological order some of the most representative planners in AP, focusing on those that represented a breakthrough in the state of the art at a given time.

- **STRIPS.** This domain-independent planner performed backward chaining in the state space without heuristics (Fikes and Nilsson, 1971). It ordered the goals and tried to satisfy iteratively each goal independently of the others. This planner set the trend for subsequent planners that used a similar approach but added domain-independent heuristics and machine learning techniques to their reasoning process, like PRODIGY (Veloso et al., 1995). Newer planners like PRODIGY also supported ADL.

- **UCPOP.** The most representative partial-order planner (Penberthy and Weld, 1992). Developed in the early '90s, it performed search in the plan space by detecting flaws such as conflicts (threats) between actions and repairing them. It also supported ADL. Additionally, the plan returned as output was a partially-ordered plan.

- **Graphplan.** It builds a graph that represents all possible parallel plans (plans in which independent actions can be executed concurrently) up to a given length n (Blum and Furst, 1997). Then, it either finds a solution of parallel length n or proves iteratively that there is no solution plan up to n and increases n by one to continue the search. The plans returned are step-optimal; that is, plans are optimal in the sense that there is no other plan with fewer parallel steps. Actually equivalent to an Iterative Deepening Depth-First Search algorithm in regression, it represented an impressive leap in performance thanks to the constraints that are propagated forward along the graph.

- **SATPlan.** It compiles iteratively the planning task into several SAT tasks using the concept of parallel steps (Kautz et al., 2006). Inspired by Graphplan, it maps the problem into a SAT instance and uses a state-of-the-art SAT solver to find a solution or prove that there was no solution up to the current horizon.

- **Heuristic Search Planner (HSP).** HSP (Bonet and Geffner, 2001) searches in the state space using Weighted A* (in which, given $s \subseteq S$, $g(s)$ denotes the cost of the current path from I to s, $h(s)$ denotes the value of some heuristic h in s, and $f(h) = g(s) + h(s)$) and h^{add}, a heuristic based on a delete-relaxation of the problem (a version of the original planning task in which all the *delete* effects of all the actions $a \in A$ are ignored). h^{add} is computed by iteratively adding the costs of achieving the preconditions of the actions that achieve the goals. The precursor of most current successful planners, it performs forward search (although backward search was also implemented).

- **Fast Forward (FF).** Similar to HSP, FF introduced several new techniques that increased its efficiency considerably (Hoffmann and Nebel, 2001). Examples of the contributions of FF are: Enforced Hill Climbing, a local search algorithm based on regular Hill Climbing that searches exhaustively until a node with better heuristic value than the last found best node is discovered, in which case the search is continued discarding the expanded nodes; the relaxed-plan heuristic, similar to h^{add} but it returns instead the number of actions of a plan that solves the delete-relaxation version of the problem;

and helpful actions, actions extracted from the relaxed plan that increase the greediness of the search algorithm.

- **Fast Downward.** Also an evolution of HSP (Helmert, 2006), it is in fact a planning framework that uses a multi-valued representation of the problem (Helmert, 2009) and implements a broad range of techniques. Among the most successful ones we can find the landmark counting heuristic, the causal graph heuristic and the use of multiple open queues to name a few. Fast Downward will be the basis of most implementations in this thesis.

2.5 Constraints, Invariants and Other Common Concepts

In many real life problems there are restrictions that are obvious to humans but that may not be trivial for machines to infer. For example, imagine a problem in which a person can move from one room to another. In this case if a proposition *p'* represents that person is at the first room and another proposition *p"* represents that person is at the second room, it is trivial for a human to deduce that *p'* and *p"* cannot be true at the same time. However since machines cannot derive any information from the semantics of the planning problem, an algorithm should derive such restrictions.

These restrictions represent constraints in the planning model, as they may reduce the valid domain of the variables of the problem. Many of these restrictions must hold in every relevant path to the goal; in this case we call them invariants of the problem.

Definition 1. *(Invariant) An invariant of a problem is a logical function that must hold in every solution of the problem.*

The scope of some invariants can be more precise: *i.e.* a state invariant is a logical function that must hold in every state along any solution path. For example, the fact that *p'* and *p"* cannot be true at the same time is a state invariant of the form $\neg(p' \wedge p'')$ because it must be true in every state that may appear in a solution path of the problem.

During search, however, states that do not satisfy some invariants of the problem may be generated. These states cannot belong to a solution path and can be safely discarded (or pruned). These states are often called spurious states.

Definition 2. *(Spurious state) A state is spurious if it cannot appear in any solution path of a problem.*

This definition differs from the original definition of spurious state presented by Bonet and Geffner (Bonet and Geffner, 2001) in that it does not depend on the initial state, and hence by our definition spurious states can be reached by applying a sequence of actions from I. This is so because in this thesis we consider more general pruning methods that employ state invariants other than the ones proposed by Bonet and Geffner. Thus, the set of detectable spurious states under this definition will be a superset of the detectable spurious states under Bonet and Geffner's definition.

Note that there is an additional alternative definition of spurious state in the abstraction literature by Zilles and Holte (2010): if *abs* is an abstraction of *P*, then an abstract state *s'* is spurious if it is reachable from *abs(I)* but it does not exist any state *s"* reachable from *I* in the original search space such that *abs(s")=s'*. This definition is tailored for abstract spaces, so we will use Definition 2 instead.

This concept can be generalized to any set of propositions, that is, any partial state. For instance, due to the instantiation of the domain and the problem specification, a planner may create actions whose set of preconditions is spurious because it violates some invariant of the problem. Such actions are never applicable nor can appear in a solution plan and are labeled as spurious actions.

In this section we will present several recurrent concepts related to constraints and invariants in AP that will be used extensively in the development of the thesis.

2.5.1 Mutual Exclusivity between Propositions

As in the example previously shown, a common form of state invariant is when one or more propositions cannot be true at the same time. This state invariant is known as a relationship of mutual exclusivity between propositions.

Definition 3. *(Mutex) A set of propositions $M = \{f_1, \ldots, f_m\}$ is a set of mutually exclusive propositions of size m (mutex of size m) if there is no state $s \subseteq S$ that may belong to a solution path such that $M \subseteq s$.*

The aforementioned example that stated that a person cannot be at two rooms at the same time is in fact a mutex of size 2 (also known as a *binary mutex*), but mutexes of greater size are also possible. An example of mutex of size 3 in *Blocksworld* is a tower of three blocks that forms a cycle, *i.e.* *M*=$\{$*(on A B), (on B C), (on C A)*$\}$. An important remark is that no subset of two elements of *M* is a mutex, which means that mutexes of greater size cannot be built parting from sets of mutexes of smaller size.

The first method proposed to infer mutexes of size 2 was the constraint propagation method of the planning graph (Blum and Furst, 1997). The mutexes obtained are in fact more general in the sense that they included the concept of time step: every *nth* proposition level of the planning graph has its own set of binary mutexes, which means that there is no state no farther than *n* parallel steps away from *I* that satisfies both propositions of the mutex. These mutexes are non-static binary mutexes, as opposed to the static mutexes of Definition 3[1].

Nevertheless, Graphplan is also able to find static mutexes too. When propagating the constraints, the planning graph is guaranteed to level off at some level.

[1]Throughout the thesis all the references to mutexes will be to static binary mutexes unless otherwise specified.

This happens when there are two consecutive proposition levels that contain the same propositions and the same mutexes (which means that from that moment on all the proposition levels will be identical). Mutexes that appear at this level are guaranteed to hold in every state no matter how far away in terms of parallel steps a state may be from *I* and are thus static.

The most common method to find mutexes is the h^m heuristic (Bonet and Geffner, 2001). h^m performs a reachability analysis in P^m (Haslum, 2009), a semi-relaxed version of the original problem in which atoms are actually sets of m propositions. This way, if the value of h^{max} of an atom in P^m is infinite (which means that it is unreachable in P^m) then we can conclude that that atom is a mutex of size m. However, the time needed to compute h^m grows exponentially with m, so in most cases it is unpractical to compute mutexes of size greater than two (in fact, increasing the size of m is a clear case of diminishing returns, as the bigger the size of the mutex is the fewer spurious states will tend to contain it and thus the less useful it is in a search algorithm to prune spurious states). h^2 returns the same static mutexes as Graphplan, although it can be implemented in a more efficient way.

h^m can be computed backwards - generally after having been computed forward (Haslum, 2008). This is done by reversing the planning problem as proposed by Massey (1999). This method finds the same static mutexes found by the reversed version of the planning graph based on Massey's reversal proposed by Pettersson (2005). Again, this means that the mutexes found by h^m depend on the direction of the reachability analysis[2] and that alternating the computation of h^m forwards and backwards might lead to finding more mutexes.

Another method for finding binary mutexes is the monotonicity analysis generally employed to generate a multi-valued formalization of the problem (Helmert, 2009). This monotonicity analysis ensures that the number of propositions true at the same time that belong to a set $M_a = \{p_0, p_1, \ldots, p_n\}$ can never increase. Hence, if the number of propositions of M_a true in the initial state is one (formally, $|M_a \cap I| = 1$), then all possible pairs of propositions of M_a are binary mutexes ($\forall q_m = \{p_i, p_j\}$ such that $p_i, p_j \in M_a \wedge p_i \neq p_j$, q_m is a binary mutex). For example, in a problem in which a person can move between three different rooms *A*, *B* and *C*, the number of propositions true in the set M_a=$\{$*(at A), (at B), (at C)*$\}$ can never increase, as whenever a person moves to a room he/she is not at the room of origin anymore. Therefore, we can infer that $\{$*(at A), (at B)*$\}$, $\{$*(at A), (at C)*$\}$ and $\{$*(at B), (at C)*$\}$ are mutexes, that is, a person cannot be at any two rooms at the same time.

Computing the monotonicity analysis is in most cases much more efficient that computing h^2 because it works exclusively with the domain definition and the initial state. However, the set of mutexes found by the monotonicity analysis is a strict

[2]This is the reason why we avoided any mention of directionality and reachability in Definition 2 and Definition 3. This is in contrast to previous works in which spurious states are defined as states unreachable from *I* because they contain mutexes computed forward, which is a subset of the cases we will analyze in this thesis.

subset of the set of mutexes found by h^2.

A final remark regarding the computation of mutexes is that the representation of the task must take into account negated propositions to obtain a more complete set of mutexes. For example, if a propositional formalization is used and $\{p, \neg p'\}$ such that $p, \neg p' \in S$ is a mutex of the problem, unless $\neg p'$ is represented explicitly this mutex will go undetected. Let's see a more tangible example: the *Matching-Blocksworld* domain is a version of the regular *Blocksworld* with more than one arm in which arms and blocks have a polarity and blocks have a related predicate *(solid ?block)* true in I for every block. When a block is put down on the table or stacked on top of another block using an arm with the wrong polarity then the block becomes non-solid, represented by $\neg$*(solid ?block)*. Once a block becomes non-solid no other block can be stacked on top of it, which means that some goals may not achievable anymore. Intuitively, if for instance there are two blocks *A* and *B* and *(on A B)* $\notin I$ and *(on A B)* $\in G$, then $\{$*(on A B)*,$\neg$*(solid B)*$\}$ is mutex. If $\neg$*(solid B)* is not explicitly represented when computing the mutexes then the mutex $\{$*(on A B)*,$\neg$*(solid B)*$\}$ will not be found, which may have a very important negative impact on the search.

2.5.2 Invariant Groups

Invariant groups are sets of propositions related to some invariant of the problem. The aforementioned monotonicity analysis was devised to find *"at-most-1"* invariant groups for their use as variables of a multi-valued formalization (Helmert, 2009). These *"at-most-1"* invariant groups can be derived from any binary mutex computation method: any set $M_a = \{f_0, f_1, \ldots, f_n\}$ such that, if $\forall p_m = \{f_i, f_j\}$, then $f_i, f_j \in M_a \wedge f_i \neq f_j$, and p_m is mutex, is an *"at-most-1"* invariant group, so *a priori* these invariant groups do not offer more information than taking into account individual binary mutexes.

Nevertheless, *"at-most-1"* invariant groups can also be used to derive stronger constraints. In particular, by performing a simple analysis of the effects of the actions one can infer that an *"at-most-1"* invariant group is in fact an *"exactly-1"* invariant group: check every action $a \in A$ and, if every action a that adds exactly one proposition $p \in M_a$ also deletes exactly one proposition $p' \in M_a$ and *vice versa*, then M_a is an *"exactly-1"* invariant group. Formally, M_a is an *"exactly-1"* invariant group if M_a is an *"at-most-1"* invariant group and $\forall a \in A :$ $|add(a) \cap M_a| = |del(a) \cap M_a|$.

"exactly-1" invariant groups are strictly more constrained than *"at-most-1"* invariant groups as one can infer an additional logical constraint from then: if $M_a = \{f_0, f_1, \ldots, f_n\}$ is an *"exactly-1"* invariant group then $(f_0 \vee f_1 \vee \ldots \vee f_n)$ is a state invariant of the problem. This constraint represents the lower bound of 1 that defines the *"exactly-1"* property of the invariant group.

Note that all the variables $v \in \mathcal{V}$ of multi-valued formalizations of problems are *"exactly-1"* invariant groups, but not all the *"exactly-1"* invariant groups of the problem need to be variables. Also note that in some problems there may not be *"exactly-1"* invariant groups or they may not be useful for their use as variables; if this is the case, regular *"at-most-1"* invariant groups can be completed with an additional fluent (represented in Fast Downward with the value $\langle$*none of those*$\rangle$ to turn them into *"exactly-1"* invariant groups (Helmert, 2006)). When this is done, no additional state invariant can be inferred, as the additional $\langle$*none of those*$\rangle$ fluent represents the constraint $\neg(f_0 \vee f_1 \vee \ldots \vee f_n)$, which would make the state invariant derived from the *"exactly-1"* invariant group evaluate to *true* in all cases.

From this point on whenever the term "invariant group" is used, we will mean *"exactly-1"* invariant group unless otherwise specified.

2.5.3 *e-deletion*

State invariants are not the only invariants of a planning problem. Certain properties of the actions, like the regular *add* and *delete* effects, are invariants that link actions and partial states. This is so because they imply that after the execution of the actions the resulting state must contain (and not contain) the added and deleted propositions respectively. These *add* and *delete* invariants can be trivially obtained from the definition of the domain, but more complex invariants can also be inferred.

An important notion for the thesis is *e-deletion* (Vidal and Geffner, 2005), an invariant related to the regular *delete* effects of the actions of the problem. The definition of *e-deletion* is as follows:

Definition 4. *An action $a \in A$ e-deletes a proposition $p \in S$ if p must be false in every state resulting from the execution of an ordered set of operators whose last operator is a.*

There are three cases that determine when a proposition $p \in S$ is *e-deleted* by an action $a \in A$:

- *a deletes p*;
- it has a set of preconditions mutex with *p* and does not add *p*;
- or it adds a set of propositions mutex with *p*;

e-deletion generalizes the regular *delete* effects in the sense that a given proposition p can be *e-deleted* by an action even if it is not mentioned in the action.[3] For example, in the *Blocksworld* domain the action *(stack b c) e-deletes (on a b)* because it adds *(clear b)*, which is mutex with *(on a b)*. This can be inferred even if *(on a b)* is not mentioned by *(stack b c)*.

In multi-valued representations, deleting a fluent *f* means changing the value of the variable $v \in \mathcal{V}$ it corresponds to, which is equivalent to adding a fluent mutex with *f*. Hence, the first case is a particular instance of the third case in multi-valued representations.

[3]Note that as per the PDDL specification regular *deletes* do not require the deleted proposition to be true. This means that a *delete* effect may not actually delete a given proposition $p \in S$ but just ensure that after the execution of the action $\neg p$ must hold.

2.5.4 Reasonable Orders between Propositions

Prior to the search phase, some orders between propositions can be inferred. Such orders are invariants if they are guaranteed to hold for every solution plan (in which case they are often referred to as *sound orders*). Most orders have been proposed in the context of landmarks (Hoffmann et al., 2004), which will be discussed at length in Section 2.6. However, reasonable orders (Koehler and Hoffmann, 2000) are relevant to all the propositions of the problem and will appear in relation to a broad set of techniques in this thesis, so they are introduced here.

Definition 5. *(Reasonable order) A proposition $a \in S$* is reasonably ordered before another proposition *$b \in S$ ($a <_r b$) if, whenever* b *is achieved before* a*, any plan must delete* b *on the way to achieving* a*, and re-achieve* b *after or at the same time as* a.

Previous works classify reasonable orders as unsound (Hoffmann et al., 2004), as not every solution plan has to achieve *a* before *b* if *b* is ever achieved. *Obedient-reasonable orders* (Hoffmann et al., 2004) are a special case of reasonable orders that arise when all previously computed reasonable orders are assumed to hold, although they will not be considered in this work.

Definition 5 does not describe how these orders are computed. Several ways of extracting reasonable orders have been proposed; we refer the reader to the works about the goal agenda (Koehler and Hoffmann, 2000) and landmarks (Hoffmann et al., 2004) for a description of the methods. We advance though that as a contribution of the thesis we will propose both a more general description and computation method in Section 3.5.3.

2.6 Landmarks and the Landmark Graph

Planning tasks define a single set of goals. However, in many planning problems humans are able to recognize facts that must be achieved at some point of the solution plan and decompose the problem accordingly. For example, if an agent must go from *A* to *B* and the areas at which *A* and *B* are located are connected by a bridge, standing at the bridge must be true at some point in every solution plan. Hence, a human would most likely first find the way from *A* to the bridge and then from the bridge to *B*. These facts are called landmarks in AP and are an important line of research, exemplified for instance by the success of landmark-based planners like LAMA (Richter and Westphal, 2010).

Landmarks were initially defined as disjunctive sets of propositions that had to be made true at some time in every solution plan (Hoffmann et al., 2004), and later on its definition was extended to include both action landmarks (Richter and Westphal, 2010) and conjunctive sets of propositions (Keyder et al., 2010). Landmarks were also formalized in a framework that relates them to abstractions and critical paths (Helmert and Domshlak, 2009), giving them a stronger theoretical background.

Landmarks can be logical formulæ over either *S* (proposition landmark) or *A* (action landmark). Next we define both kinds of landmarks.

Definition 6. *(Proposition landmark) A proposition landmark is a logical formula over either* S *such that at least one state along every solution path must satisfy it.*

Definition 7. *(Action landmark) An action landmark is a logical formula over* A *(action landmark) such that every solution plan must satisfy it.*

L is the set of discovered landmarks of the problem. The achiever of a proposition landmark $l \in S$ is an action $a \in A$ such that $l \in add(a)$. Informally, an achiever is an action that makes a given landmark true, or *adds* it. A late achiever $a \in A$ is a regular achiever of a landmark $l \in S$ with the special condition that, in every plan executed from I, it is not possible for a to appear before l has been achieved at least once by some other action $a' \in A$.

Although finding the complete set of landmarks is PSPACE-complete (Hoffmann et al., 2004), current methods can efficiently compute a subset of the landmarks using a delete-relaxation of the problem.[4] Also and by definition all the propositions in I and G are landmarks. Partial orders between proposition landmarks can be obtained with these techniques too, which are used to build the landmark graph. The landmark graph is an important part of many of the techniques that exploit landmarks, as the interactions and orders between landmarks are often as important as the landmarks themselves. Orders between landmarks are relations between two proposition landmarks that represent the partial order in which they must be achieved. The landmark graph is a directed graph composed by the proposition landmarks of a problem and the orders between them (Hoffmann et al., 2004). There are the following orders:

- Natural order: A proposition *a is naturally ordered before b* ($a <_{nat} b$) if *a* must be true at some time before *b* is achieved
- Necessary order: A proposition *a is necessarily ordered before b* ($a <_{n} b$) if *a* must be true one step before *b* is achieved
- Greedy-necessary order: A proposition *a is greedy-necessarily ordered before b* ($a <_{gn} b$) if *a* must be true one step before *b* when *b* is first achieved

These orders are hierarchically related: necessary orders are greedy-necessary orders, which in turn are natural orders, but not *vice versa*. Reasonable orders between landmarks can also be established and are often included in the landmark graph as an additional edge. There is a slight difference in the landmark literature though: reasonable orders between landmarks are computed taking into account only the first time the landmarks are achieved. This is motivated by the fact that landmark computation methods can only ensure that landmarks are needed at least

[4]For a comprehensive review of the computation methods we refer the reader to Richter's thesis (2010).

once - as they are based on a delete-effect relaxation - even if they have to be achieved more than once in every solution plan. Also note that, unlike a relaxed planning graph, the landmark graph is not acyclic, as necessary and reasonable orders can induce cycles.

To illustrate how landmarks and the landmark graph work we will include the running example from Richter's thesis (2010). Figure 2.2 shows a planning problem in a domain similar to *Logistics* in which there are also planes that can move between locations with airport. In this problem a package must be carried from location *B* to location *E*. Round locations are regular locations and square locations are those that have an airport. Edges represent roads that can be traversed by trucks. Planes can move from any airport to any other airport. In this case an optimal plan is: *(move t D B)*, *(load o t B)*, *(move t B C)*, *(unload o t C)*, *(move p E C)*, *(load o p C)*, *(move p C E)*, *(unload o p E)*. Taking a quick look at the planning task most humans will be able to realize that some propositions must be achieved in every solution plan to solve the problem. For instance, the package is initially at a location that is not connected by ground to the destination location, so a plane must be used to carry the package and hence *(in o p)* must be true at some point. Similarly, the initial location of the package is not an airport location, so a truck must be used to carry it to some airport location - which means that *(at t B)*, *(in o t)*, *(at o C)* and *(at t C)* are landmarks too.

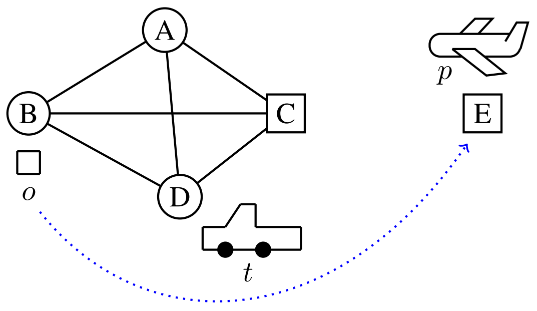


Figure 2.2: Example of a planning task with landmarks.

The landmark graph that corresponds to the example is shown in Figure 2.3. We have taken out the initial positions of the truck and the plane for simplicity. Here we can see that the all the aforementioned propositions plus a few more that we did not mention are indeed landmarks.

The edges that appear in the graph represent the orders between landmarks. All represented orders are natural orders except for one reasonable order represented by a dashed edge. In this example we can see that the orders of the landmark graph induce a total order on when the landmarks must be achieved similar to the one induced by the previously presented solution plan. This shows the potential of the landmark graph, as a partitioning of the problem based on a total order extracted

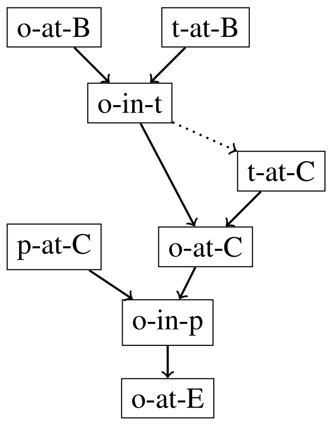


Figure 2.3: Partial landmark graph of the previous task. Regular edges represent natural orders, dashed edges represent reasonable orders.

from the landmark graph would be very helpful for the search in this case. Next we present an overall summary of the most common approaches that exploit the information provided by landmarks and the landmark graph.

2.6.1 Partitioning of the Problem using Disjunctive Landmarks

The first proposed way of using landmarks was partitioning the problem into several smaller ones (Hoffmann et al., 2004). This has the advantage of potentially obtaining an exponential gain by reducing the depth of the problem. It works by taking the leaves (landmarks not preceded by other unachieved landmarks) of an acyclic landmark graph and turning them into a disjunctive set of goals. When this goal is achieved by using any regular planner, the landmark graph is updated and the search begins from the last state with a new disjunctive goal until all landmarks have been achieved.

Although this achieves important speedups in some cases, it also has some shortcomings. First, the total order is guessed in a rather random way, as the search will almost always achieve the closest landmark even if others belonging to the disjunctive goal must come first. Second, this method must eliminate cycles from the landmark graph, which may lead to an important loss of information. Third, the increased greediness of the approach may lead to dead ends or may make the algorithm commit to unpromising areas of the search space. Because of this this approach has obtained worse results overall than just using the original goal set and a state-of-the-art planner.

2.6.2 Landmark Counting Heuristics

Since all the landmarks in L must be achieved at some point, counting how many (disjunctive) landmarks have not been achieved yet can be used as an estimation of the distance to the goal. The first time this was proposed was as an unadmissible heuristic to be used in a forward heuristic planner, LAMA (Richter and Westphal, 2010), which proved to be very effective in combination with other techniques already implemented in Fast Downward. At every state LAMA keeps track of which landmarks have not been achieved and which landmarks are required again, and returns the cost of achieving those landmarks. Furthermore, it computes a relaxed plan to the closest disjunctive set of unachieved - or required again - landmarks to extract preferred operators.

An admissible version of LAMA's heuristic was later proposed. The admissible landmark counting heuristic h_{LA} formulated by Karpas and Domshlak (2009) is an admissible estimation of the distance to the goal computed by adding the cost of achieving the propositional landmarks that are still needed using a cost-partitioning scheme to ensure admissibility. There are two versions of h_{LA}. The simplest version splits uniformly the cost of every achiever amongst the landmarks that they achieve (h_{LA}^{uni}), so the cost of each landmark is the minimum of those "split costs". The more complex version solves a Linear Programing problem per state that yields the maximum cost of achieving the required landmarks by selecting which achiever contributes to the cost of which landmark and by how much while keeping the estimate admissible (h_{LA}^{opt}).

The value returned by h_{LA} depends on which landmarks are true in a given state $s \in S$ and on which landmarks have already been achieved. The latter depends on the path or paths that led from I to s, so this information must also be encoded. This makes h_{LA} a *path-dependent* heuristic, which means that h_{LA} is inconsistent and may yield different values for the same state s. Additionally, as the landmarks are computed from a delete-relaxation of the problem, h_{LA} is bounded by h^+, the cost of the optimal plan of the delete-relaxation of the problem.

A more recent heuristic based on the concept of landmarks is the landmark-cut heuristic h^{lmcut} (Helmert and Domshlak, 2009). Unlike the previous heuristics it is not computed using propositional landmarks nor the landmark graph and requires no previous precomputation. Instead it builds a justification graph similar to a relaxed planning graph and iteratively computes cuts in the justification graph that separate I from G. These cuts are disjunctive sets of actions, which are in fact disjunctive action landmarks. Again, h^{lmcut} is bounded by h^+ for the same reason as h_{LA}.

Part II

Objectives

As presented in the introduction, this thesis consists of three main lines of research from the point of view of classical planning. Although these three lines of research correspond to the structure of the document, the main objectives of the thesis mark the development and the flow of the study as a whole. Already stated in the title and in the abstract, the main objective of this thesis is to study novel ways of generating and exploiting intermediate goals for their use in classical planning. For that, a series of smaller objectives must be set, objectives which will determine the steps to be taken throughout the development of the thesis.

The first step is analyzing how to generate intermediate goals given a particular technique, as it seems more logical to begin from a known technique and use it for our purpose. In this case, regression is the technique of choice. However, regression has arguably fallen out of fashion in the planning community, which means that the current state of the art in regression must be revised. Therefore, the very first step of this thesis is to study of the viability of regression in planning, analyzing the flaws of common regression techniques and proposing updated alternatives that aim to bridge the gap between regression and other state-of-the-art approaches.

Once the major issues in regression are identified and solved, the next step is employing regression to actually generate intermediate goals. The way regression can be used for this is unclear, so we will consider several alternatives and propose techniques that may seem promising in terms of efficiency. Most conclusions extracted from the prior study of regression will probably prove to be essential in this phase.

After employing regression to generate intermediate goals, we want to extrapolate the generation of intermediate goals to a more general case. In particular, we want to analyze the possibility of obtaining intermediate goals at random, choosing states by sampling uniformly the search space instead of depending on regression. There are several important problems with this that must be overcome, as the search space of a planning task is implicit and there is no guarantee that a randomly sample state will belong to the search space. To ensure that the sampled state is useful we analyze whether the lessons learnt in regression apply to the more general case of random sampling, and, if they do not suffice, propose novel techniques to solve this issue. Along with the study of the generation of random intermediate goals, we will also propose search algorithms that exploit them to hopefully obtain better results than current state-of-the-art search algorithms.

The last objective of the thesis is to employ the knowledge obtained from the analysis of the generation of intermediate goals in the field of landmarks. Landmarks are intermediate goals by definition, so we want to discover which of our findings apply to them. These is a twofold task, as it aims to provide a better understanding of landmarks and to find new landmark-based ways of generating intermediate goals.

In summary, these are the objectives of the thesis in order:

- Analyze the shortcomings of regression and find ways of addressing them.

- After ensuring that the use of regression is viable, propose techniques to generate and exploit intermediate goals based on regression.
- Once we have a better understanding of how regression and intermediate goals work together, generalize to random sampling of states.
- If we find that random sampling of states is a valid way of generating intermediate goals, propose a search algorithm able to benefit from them to overcome some of the problems of current state-of-the-art search algorithms.
- Extrapolate the findings from random sampling to landmarks, and study whereas they can be used to enhance the usability of landmarks.

In the conclusions we will check whether these objectives have been fulfilled, pointing out our major discoveries and putting in context the theoretical and empirical results obtained along the development of the thesis.

Part III

Regression, Invariants in Symbolic Regression and Backward Generated Goals

Chapter 3

Revisiting Regression in Planning

Heuristic search with reachability-based heuristics is arguably the most successful paradigm in Automated Planning to date. In its earlier stages of development, heuristic search was proposed as both forward and backward search. Due to the inherent disadvantages of backward search, during the last decade researchers focused mainly on forward search, and backward search was abandoned for the most part as a valid alternative. In the last years, important advancements regarding both the theoretical understanding and the performance of heuristic search have been achieved, applied mainly to forward search planners. Motivated by the advances in forward heuristic search we revisit regression in planning with reachability-based heuristics, trying to extrapolate to backward search current lines of research that were not as well understood as they are now.

3.1 Introduction

Automated planning is the task of finding a sequence of actions that, from a given initial state, reaches a set of goals. It is one of the oldest areas of research in AI, but only relative recently domain-independent planners have been able to solve non-trivial tasks. Heuristic search (Bonet and Geffner, 2001) has become one of the most important paradigms in the area, as shown by the results of the most recent International Planning Competitions (IPC)[1] and the much higher number of publications in AP about heuristic search compared to publications about other successful approaches like local-search (Gerevini and Serina, 2002) or SAT-based planning (Kautz et al., 2006; Rintanen, 2010).

Search can be done either advancing from the initial state towards a goal state, called forward search or progression, or from a goal state to the initial state, called backward search or regression. In heuristic search most works use combinatorial

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

domains as benchmarks. Combinatorial domains are in most cases trivially invertible and have a single goal state. Examples of this kind of benchmarks are the *n-puzzle* and the Rubik's Cube. Therefore, often forward and backward search are analogous and the same techniques and conclusions are relevant to both cases. However, this is not true in AP. While the initial state is completely defined due to the closed-world assumption, goals are defined as partial states. Thus, the value of propositions (or variables in a multi-valued formalizations) not included in the goal set are unknown and multiple goal states are possible. For example, in the *Blocksworld* domain the goal set includes most of the time only propositions derived from the predicate *"on"*, like *(on a b)* and *(on b c)*, which means that the value of other propositions, like *(arm-empty)* and *(clear a)*, is unknown. As a consequence, backward search reasons over sets of facts and not complete states, making the search more akin to state-set search (Pang and Holte, 2011) than to traditional heuristic search.

Due to these differences, backward search planners have several drawbacks. The most important ones are:

- techniques developed for forward search may not be (trivially) applicable or useful in regression;
- duplicate detection is more complex due to partial states; and
- spurious states may be generated.

In fact, early research on heuristic search in AP studied both forward and backward search (Bonet and Geffner, 2001), but these drawbacks lead to a worse performance of regression planners compared to progression planners. For this reason, research on backward search in AP was discontinued in many cases.

Over the last decade important works shed light on both the formal properties of forward search, like the relationship between different heuristics (Helmert and Geffner, 2008; Helmert and Domshlak, 2009), and the empirical impact of widely used techniques, like preferred operators and deferred heuristic evaluation (Richter and Helmert, 2009). Nevertheless and despite the strong relationship between forward and backward search, this has not been exploited to improve over the results of previous backward search planners. In particular, some techniques that may address the shortcomings of regression in satisficing planning, like preferred operators, seem to have potential. In this section we analyze how efficient techniques commonly used in progression can be extrapolated to backward search. We define some new concepts related with regression and propose several novel techniques implemented in a new regression planner, based on Fast Downward (Helmert, 2006), called FDr (Fast Downward Regression). In particular, the concepts and techniques are the following:

- formalization of regression in SAS$^+$;
- disambiguation of states and action preconditions;

- action pruning using e-deletion;
- improvements on the computation of the applicable actions;
- novel definition of reasonable orders;
- computation of reachability heuristics and preferred operators in regression;
- usage of P^m formulations of the planning task to infer more informed heuristics in regression; and
- computation of strong precedences related to the context-enhanced additive heuristic in regression.

Experimental results show that the techniques analyzed are in fact useful for regression, leading to the development of a regression planner consistently more efficient than its predecessors.

3.2 Planning Formalizations in Regression

Regression search in planning starts from the goals and applies the actions backwards to find a path from the set of goals G to the initial state I. The set of goals G in a planning task constitutes a partial state, as the value of the propositions not mentioned in the goals $S \setminus G$ is unknown when a propositional formalization is used. This also means that the closed world assumption cannot be enforced due to the existence of undefined values.

Action preconditions are also partial definitions of the states in which the actions can be applied. Thus, states generated by regression are partial states too. This is so because, if $a \in A$ is an action of the problem, the value of the propositions *eff(a)* (that is, those propositions mentioned in the effects of the action) prior to its execution may not be inferred. For instance, applying in regression an action $a \in A$ *(move A B)* such that *pre(a)* = $\{$*(at-agent A)*$\}$ and *eff(a)* = $\{$*(at-agent B),*$\neg$*(at-agent A)*$\}$ means that the value of $\{$*(at-agent B)*$\}$ after applying a in regression is unknown, as its value is not defined in *pre(a)* and it was added by a. The underlying reason is that $\{$*(at-agent B)*$\}$ may have been in theory true or false prior to the execution of a, because *(at-agent B)* $\in$ *eff(a)* does not imply that $\neg$*(at-agent B)* $\in$ *pre(a)*.

Our formal definition of a propositional planning task in regression is as follows: given a propositional planning task *P=(S,A,I,G)*, the definition of *P* for regression is a tuple *P'=(S,A,I',S_G)* where *I'*= *G* is the initial (partial) state; S_G is the set of goal states $S_G = \{s \mid s \subseteq I\}$.

The set of actions *A* is the same as in progression, but their applicability is redefined as follows: $a \in A$ is applicable in a partial state $s \subseteq S$ if it is *relevant* ($add(a) \cap s \neq \emptyset$) and *consistent* ($del(a) \cap s = \emptyset$). An action *relevant* to a set of propositions is also known as a *supporter* of the set of propositions. The resulting state $s_r \subseteq S$ obtained from regressing a in s is $s_r = (s \setminus add(a)) \cup pre(a)$. This can also be seen as a reversed action a' in which $add(a)$ is a set of disjunctive preconditions of a', $del(a)$ are negative preconditions of a' and $pre(a)$ are the *adds* of a'.

As picturing regression may be in some cases complicated, here we present a very simple example. In Figure 3.1 we have a set of goal propositions and two supporting actions.

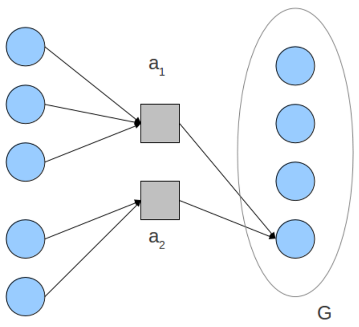


Figure 3.1: Original goal G and and two actions $a_1, a_2 \in A$ supporters of G.

After applying a_1 we have a new state s', in which the proposition added by a_1 does not appear but which contains the preconditions of a_1. This is shown in Figure 3.2.

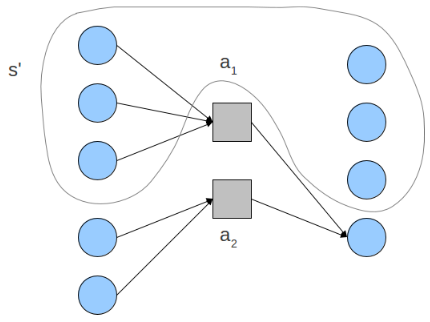


Figure 3.2: Resulting state $s' \subseteq S$ obtained by applying a_1 in G in regression.

Progression and regression in planning are not symmetric as each state in the regression state space may represent a set of states of the progression state space. Also, states reachable in progression may not be reachable in regression and *vice versa*. Such states are always spurious.

A special case that occurs when reasoning with partial states is subsumption of states:

Definition 8. *(Subsumption of states) Given two states $s_i, s_j \subseteq S$, s_i subsumes s_j ($s_i \sqsubseteq s_j$) if $s_i \subseteq s_j$ and $g(s_i) <= g(s_j)$, where $g(s_i)$ and $g(s_j)$ are the best cost among all the visited paths that lead to s_i and s_j respectively.*

Subsumption of states changes how duplicate detection should be applied in regression. In progression, a state $s_j \subseteq S$ can be pruned if a state $s_i \subseteq S | s_i = s_j \wedge f(s_i) <= f(s_j)$ has been already generated. This is easily detectable in constant time if a structure such a hash table is used, as states in progression are completely defined. In regression duplicate detection can be extended to the case in which $s_i \sqsubseteq s_j$, because achieving s_i is strictly less costly than achieving s_j. However, $s_i \subseteq s_j$, a necessary condition for $s_i \sqsubseteq s_j$, is not detectable using a hash function. This means that complete duplicate detection is not possible by using only a hash table, which can have an important negative impact in the performance of the search.

3.2.1 Regression in SAS$^+$

Although regression had been formally defined for propositional tasks, only one work by Rintanen (2008) went beyond *STRIPS* in terms of formal definitions of regression. Because of this, here we address the formal definition of regression in SAS$^+$.

Given the SAS$^+$ formalization of a planning task in progression as Π=($\mathcal{V}$,s_0,$S_\star$,$\mathcal{O}$), a SAS$^+$ task in regression is a tuple Π'=($\mathcal{V}$,s'_0,$S'_\star$,$\mathcal{O}$), where $s'_0 = s_\star$ and $S'_\star$ is the set of partial states subsumed by s_0, following the next definition of subsumption in SAS$^+$.

Definition 9. *(Subsumption of states in SAS$^+$) Given two SAS$^+$ states s_i and s_j, s_i subsumes s_j ($s_i \sqsubseteq s_j$) if $\forall v \in \mathcal{V}$, $s_i[v] = s_j[v] \vee s_i[v] = \mathbf{u}$.*

Applicability of actions in SAS$^+$ also requires a new definition.

Definition 10. *(Applicability of actions in SAS$^+$) An action $o \in \mathcal{O}$ is applicable in a partial state s in regression if $\forall v_i \in V : s[v_i] \neq \mathbf{u}$, if $post(o)[v_i] \neq \mathbf{u}$ then $s[v_i] = post(o)[v_i]$; and if $prev(o)[v_i] \neq \mathbf{u}$ then $s[v_i] = prev(o)[v_i]$ (o is* consistent *with s) and $\exists v_i \in V : s[v_i] = post(o)[v_i] \wedge s[v_i] \neq \mathbf{u}$ (o is* relevant *to s). The resulting state s' obtained from applying o in s in regression is equal to s except that $\forall v_i \in \mathcal{V}$ s.t. $pre(o)[v_i] \neq \mathbf{u} : s'[v_i] = pre(o)[v_i]$ and $\forall v_i \in \mathcal{V}$ s.t. $prev(o)[v_i] \neq \mathbf{u} : s'[v_i] = prev(o)[v_i]$.*

Note that applicability of actions in SAS$^+$ is more restricted than in a propositional representation, as prevail preconditions must be taken into account when checking the consistency of an action. In fact, the set of applicable actions in regression in P' is a subset of the applicable actions in regression in Π' for any partial state. All the actions applicable in a propositional representation that are not applicable in SAS$^+$ lead to spurious states.

Theorem 1. *Let P' be a planning task in a propositional formalization and Π' its SAS$^+$ equivalent, $s, s' \subseteq S$ be partial states, and $a \in A$ be an action in P' equivalent to an operator $o \in \mathcal{O}$ in Π'. If for any given partial state s a is applicable in regression but o is not, then the state s' generated by applying a in regression in s is spurious.*

Proof. Given a variable $v_i \in \mathcal{V}$, let $s'[v_i] = x : x \in D_{v_i}$ be a fluent that corresponds to a proposition $p \in P$ such that $s'[v_i] \neq s[v_i]$, $s'[v_i] \in prev(o)$ and $p \in pre(a)$. Since $s'[v_i] \neq s[v_i]$, o is not applicable in regression in s as per Definition 10.

Assume that a is applicable in regression in s; then, the resulting state s' will contain both $s'[v_i] = p$ and $s[v_i]$. As $s'[v_i]$ and $s[v_i]$ belong to the same variable v_i, $s'[v_i]$ and $s[v_i]$ are mutex. Therefore, s' is spurious. □

Partial states in SAS$^+$ require the use of the undefined value **u** so the closed world assumption is not dropped. SAS$^+$ allows it by adding it to the domain of all the variables $v \in \mathcal{V}$ in a preprocessing step. Hence, any SAS$^+$ planner can do regression with no changes in their search algorithm apart from modifying the applicability and effects of actions. Identical partial states would be detected as duplicates, but subsumption of states would not.

3.3 Regression and Backward Heuristic Search: HSPr

As stated in the introduction, the first heuristic search planner that worked in progression, HSP (Bonet and Geffner, 2001), had a version that searched in regression, HSPr. HSPr uses a propositional formalization, a weighted A* algorithm and the additive heuristic, h^{add} (Bonet and Geffner, 2001). h^{add} is the sum of the accumulated cost of achieving every goal proposition in a delete-free version of the problem. This heuristic is also the one that HSP uses, but there is a crucial difference: h^{add} is computed by doing a reachability analysis whose source is either the initial state I (in regression) or the evaluated state (in progression) and which computes the accumulated cost to every proposition $p \in S$ that has been explored during the computation of the heuristic. As HSP does progression, the source of the reachability analysis changes every time a state is evaluated and thus the reachability analysis must be computed per state. However, HSPr uses the initial state I as the source of the reachability analysis, so it can avoid repeating the reachability analysis by caching h^{add} for all the propositions of the problem and evaluate a state in a fraction of the time HSP needs. This speed up in the computation of h^{add} is critical, as the bottleneck in heuristic search planners is most of the time the computation of the heuristic (Liu et al., 2002).

The much higher rate of generation of states of HSPr is the main advantage of HSPr over HSP, but HSPr had several important disadvantages too. To deal with the frequent generation of spurious states in regression, HSPr uses mutexes found with h^2 to prune them. After a state $s \subseteq S$ has been generated, HSPr checks if it contains two propositions $p_i, p_j \in S$ such that $\{p_i, p_j\}$ are mutex and, if that is the case, discards it. However, mutexes may not suffice to prune spurious states in some domains, which may lead to an exponential increase in the number of generated states in regression.

Second, when using a propositional representation, detection of duplicate states requires modifications in the algorithm due to the presence of propositions whose value may be unknown. Furthermore, regular duplicate detection using hash functions is unable to detect subsumption of states, which again may lead to an exponential increase in the number of generated states.

Finally, additive schemes in the computation of the heuristic may cause inaccuracies when variables that were undefined in the initial state in regression I' are assigned a value, suddenly adding their cost to the partial state. Actually this is extensible to all the partial states: the cost of unknown propositions is not accounted for when computing the heuristic, so h^{add} may differ greatly depending on the number of unknown propositions. This causes anomalies in the behaviour of the search algorithm, leading sometimes to the exploration of h plateaux.

For these reasons HSPr performed overall worse than HSP. These problems combined with the success of some techniques introduced by other forward heuristic planners, like Fast Forward and Fast Downward, made researchers abandon for the most part backward search in planning.

3.4 Revisiting Regression in Planning

In this section we revisit some important concepts originally defined for forward search, adapting them to their use in regression. Each of the subsections describes the set of techniques, proposes a novel implementation in regression and defines them formally if needed.

3.4.1 Invariant Groups and Disambiguation

In partial states, the value of some propositions may be unknown. Nevertheless, the value of other known propositions may allow inferring whether a proposition $p \in S$ is true or false (or the value of a variable $v \in \mathcal{V}$ in a multi-valued representation). We call this process *disambiguation* of a partial state.

Definition 11. *(Disambiguation) Disambiguation is the process of reducing the valid domain of the invariant groups of the problem given a set of known propositions and a set of constraints.*

Disambiguation means solving a Constraint Satisfaction Problem (CSP) in which the variables are the invariant groups of the problem, the domain of the variables is the set of propositions that conform the invariant group and the constraints are the mutexes between propositions. The set of known propositions are the propositions true in the partial state; these propositions satisfy the invariant groups they appear in and may reduce the valid domain of the rest of the invariant groups.

There are three relevant cases for any given invariant group θ for which there was no previously known $p \in S$ such that $p \in \theta$:

- only one proposition $p \in \theta$ can be true in the partial state;

- no proposition $p \in \theta$ can be true; and
- more than one proposition $p \in \theta$ can be true.

The first case means that p must be true in the state and so it can be added to it to complete it. The second case means that the CSP has no solution and that the partial state is thus spurious, as it violates the invariant that enforces that in every non-spurious state exactly one proposition $p' \in \theta$ must be true. The third case means that no additional information can be inferred.

Let's look at an example of a multi-valued planning task in which the only invariant groups are the variables in $\mathcal{V}$. Given $\mathcal{V} = \{v_0, v_1, v_2\}$ and $D_{v_0} = \{a, b, c\}$, if in a given partial state $s : s[v_0] = u$, $s[v_1]$ is mutex with $v_0 = a$ and $s[v_2]$ is mutex with $v_0 = b$, then we can infer that $s[v_0] = c$, because c is the only possible value for v_0 that satisfies the constraints. If additionally $s[v_1]$ or $s[v_2]$ are mutex with $s[v_0] = c$, then v_0 has no possible value and the state is spurious.

Disambiguation of partial states fulfills two purposes: adding information to partial states by reducing the number of unknown variables upon generation, and pruning spurious states that are undetectable by using only binary static mutexes. Having fewer unknown variables impacts the performance in two ways. The advantages of having more complete partial states is that heuristics tend to be more accurate, as the cost of fluents that otherwise would be ignored is accounted for (Domshlak et al., 2012; Bonet, 2013), and that it reduces the number of cases in which there is subsumption of expanded states thanks to cases in which $s_i, s_j \subseteq S$, $s_i \sqsubseteq s_j$ before disambiguation and $s_i = s_j$ after disambiguation.

To better understand which kind of spurious states that do not violate mutexes may be generated in regression we will provide an example. Figure 3.3 shows an example of pruning by disambiguation in the *floortile* domain.

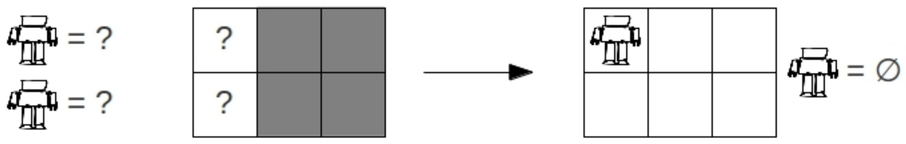


Figure 3.3: Pruning a spurious state in *floortile* by disambiguation. The second robot in the state at the right has no valid location because all cells are clear or occupied.

Problems in *floortile* consist of two or more robots that have to paint the cells of a grid. The initial state contains the locations where the robots are. The goal contains the painted cells. It is possible to find a plan doing regression in which a single robot traverses and paints the whole grid. This means that there may be a partial state in which all the cells are either painted, clear or occupied by the first robot. When disambiguating the state, we can see that there are no legal values for the variables that represent the position of the other robots, as a robot cannot be at a painted cell, a clear cell or a cell occupied by another robot. Such a state can be safely pruned even though binary mutexes with no disambiguation are ineffective to detect this case in regression.

Disambiguation of Preconditions and Spurious Operators

Partial states are not the only set of propositions that may benefit from disambiguation. In fact, Definition 11 refers to any set of propositions and not only to partial states. The preconditions of any action $o \in \mathcal{O}$ are another case in which disambiguation may be beneficial. The main benefit is that more preconditions lead to fewer unknown propositions (or variables) when generating partial states in regression. From the point of view of the disambiguation process, this means that the generated state will be partially disambiguated already. This reduces the time spent disambiguating the state if disambiguation is done at every state during search and allows obtaining some of the benefits of disambiguation of states if it is not made on a *per-state* basis. An additional advantage of having more preconditions is that heuristics may be more informed, as the model used to derive the heuristic (the delete relaxation of the problem for h^{add} and the Linear Programming model for h^{SEQ} (Bonet, 2013), for example) would be more tightly constrained. A simple example of disambiguation of preconditions is the operator *(put-down A)* in the *Blocksworld* domain, which initially only has *(holding A)* as precondition. If disambiguation is performed during the preprocessing step, additional preconditions such as $\neg$ *(arm-empty)* can be inferred.

Also, it is possible that during the grounding phase operators whose set of preconditions is spurious are generated. For example, when Fast Downward parses an operator with a negated fluent $\neg f$ as precondition, it splits the operator into several ones such that, if $f \in D_v$ given $v \in \mathcal{V}$, $\forall f' \in D_v \mid f' \neq f$ there will be an operator $o \in \mathcal{O}$ such that f' is a precondition of o. When FD splits the operator it does not check whether the new set of preconditions is not spurious, so doing disambiguation will allow to detect such cases. Operators with spurious sets of preconditions are called *spurious operators* and can be safely pruned.

An important remark is that, even though spurious operators are never applicable in progression, they may be applicable in regression (in which case they will generate spurious states) and also may affect the value of the heuristics employed by the planner. Hence, it is advisable to disambiguate operators in the preprocessing step if only to discard such operators.

3.4.2 Enhanced Applicability in Regression and Decision Trees for Successor Generation

Thanks to the caching of h^{add}, HSPr is able to evaluate states at a much faster rate than HSP. However, many generated states are spurious and are pruned using mutexes anyway, so often no performance gain is achieved. Consider for example the *Blocksworld* domain: if there is a tower of blocks and *(arm-empty)* is true in the partial state in regression, half of the actions of the domain are applicable because they add *(arm-empty)*, whereas only one (stacking the upper block) leads to a non-spurious state. The other applicable actions produce states that contain mutexes. For example, if *(on A B)* and *(on B C)* are true and *(stack B C)* is applied in regression, the generated state would contain the mutex $\{$*(on A B), (holding B)*$\}$, as *(holding B)* is a precondition of *(stack B C)*.

This problem comes from the current definition of applicability of actions in regression. In fact, the current definition makes planners knowingly generate spurious states that will be pruned afterwards using mutexes. This is a waste of effort that can be avoided using *e-deletion* instead of just regular *deletes*. The applicability of an action in regression can be modified to reflect this by ensuring that the action does not *e-delete* any fluent true in the partial state.

Definition 12. *(Consistency using e-deletion) If* $e\text{-}del(a)$ *is the set of fluents e-deleted by action* a*,* a *is consistent with partial state* s *if* $e\text{-}del(a) \cap s = \emptyset$*.*

If the novel definition of *consistency* presented in Definition 12 is used, it is guaranteed that no spurious state (detectable by using the same set of mutexes employed to infer *e-deletion*) will be generated in regression.

Theorem 2. *Let* $s \subseteq S$ *be a partial state that does not violate any binary mutex. Let* $a \in A$ *an action consistent using e-deletion with* s*. Let* $s' \subseteq S$ *be the partial state obtained by applying* a *in regression to* s*. Then,* s' *does not violate any binary mutex.*

Proof. If $s' \subseteq S$ was obtained by applying a in regression to s, then $s' \setminus s = \text{pre}(a)$ and so $(s' \cap s) \cup \text{pre}(a) = s'$. As $(s' \cap s)$ nor $\text{pre}(a)$ violate any mutex, then a mutex can only be violated if some $p \in (s' \cap s)$ and some $p' \in \text{pre}(a)$ are mutex. However, a is consistent using e-deletion with s and so $(s' \cap s) \cup \text{pre}(a)$ does not violate any mutex, which means that s' does not violate any mutex. □

This way, using *e-deletion* for consistency avoids the generation of states that violate mutexes altogether, which may translate in an important increase in efficiency - both because the spurious state is not generated and because there is no need to check whether a mutex has been violated for both spurious and non-spurious states.

This novel definition of consistency has a drawback though: it imposes an additional complexity when checking applicability, as a very high number of propositions may be *e-deleted* by an action compared to the number of regular *deletes* the action may have. For example, a regular *move* operator *e-deletes* the moving agent at every location other than the destination, which may be expensive to check. To avoid this overhead during search, precomputed decision trees as successor generators (Helmert, 2006) can be used.

Fast Downward already uses decision trees based on partial-match tries (Rivest, 1974) to avoid checking the applicability of all the actions of the problem. In Fast Downward, every inner node represents a variable $v \in \mathcal{V}$ and has an edge for each $d \in D_v$ plus the *don't care* value. Each edge leads to another inner node representing the next variable or to a leaf node. Leaf nodes contain the sets of actions that are applicable on that node.

When building the leaf (action) nodes, an action a is propagated at every inner node down the edge that corresponds to $v = d$ if $v = d \in pre(a) \vee v = d \in prev(a)$. Otherwise a is propagated down the *don't care* edge. So, the path from the root to a leaf node represents the preconditions of the corresponding action. Every action occurs in at most one leaf node, so the number of leaf nodes is bounded by the number of actions of the problem. When computing the applicable actions of a state s, at every inner node the algorithm follows the *don't care* edge and the edge that corresponds to $s[v]$. The union of the sets of actions of the visited leaf nodes is the set of applicable actions.

The main difference with the decision trees used by Fast Downward is that in regression one must deal with disjunctive preconditions, negative preconditions and unknown variables. This means that using variables from $\mathcal{V}$ in the inner nodes is not possible, as it may be possible for an action to occur in several different leaf nodes. Imagine for example an action that *e-deletes* a fluent $f \in D_v \mid v \in \mathcal{V}$; this means that that action should be propagated down every edge that corresponds to every fluent $f' \in D_v \mid f' \neq f$, which corresponds with the negative case $\neg f$. Losing this property means that there is no lower bound on the number of leaf nodes, which could make the tree grow to an exponential size.

To avoid this, the inner nodes of the decision trees we build for regression represent fluents instead of variables, similar to the propositional case. They have three edges for the cases $\top$, $\bot$ and *don't care* through which actions that add, *e-delete* and do not change the variable of the fluent are respectively propagated. If the fluent is unknown in a partial state, all the three children must be explored, as opposed to just either $\top$ or $\bot$ and *don't care* ($\star$). This combines the efficiency of decision trees in progression and the greater expressiveness required in regression.

Figure 3.4 shows a simple example of a decision tree as successor generator. As described before inner nodes have at most three edges for the cases $\top$, $\bot$ and *don't care*. However, if no action is propagated down some edge, that edge needs not to be explored (in fact, in this example only p_1 has the three edges). Note that a single leaf node can have several actions, but no action can appear in more than one leaf node.

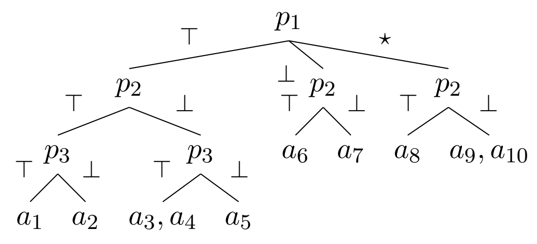


Figure 3.4: Decision tree used as a successor generator.

3.5 Reachability Heuristics in Regression

In this section, we describe how to compute recent state-of-the-art heuristics in regression, exploiting advantages of regression like caching schemes. We use a propositional formalization of the task as it is the *de facto* implementation even in planners that use a multi-valued formalization, like Fast Downward.

3.5.1 Best Supporter Caching

Most forward planners, like FF and FD, use heuristics based on a delete-relaxation of the problem. The most representative heuristics are h^{add} and the FF heuristic (Hoffmann and Nebel, 2001) - also known as the *relaxed plan* heuristic. In fact, there are multiple heuristics of this kind (Keyder and Geffner, 2009b), although the implementation of HSP includes only h^{add} because that was the only reachability heuristic developed at the time when the works on HSPr were published.

Formally, the relationship goes deeper. In fact, delete-relaxation and critical path heuristics like h^m belong to the same family of heuristics (Helmert and Domshlak, 2009). They differ mainly in the functions used to (1) select the best supporter of propositions, (2) aggregate the supporters of preconditions (subgoals) and goals, and (3) compute a cost estimation from a set of aggregated supporters (Fuentetaja et al., 2009).

The way these heuristics are computed is in fact very important for the backward case. A point in common for these heuristics is that supporters and costs are derived from a forward reachability analysis that depends exclusively on the source partial state. If the source state does not change, all the necessary information can be cached. This allows planners to compute these heuristics in regression without performing additional reachability analysis, which greatly speeds up their computation. This is exactly what HSPr does when caching costs to speed up the computation of h^{add}, although there is no reason why caching should be limited to the costs of the propositions of the problem.

As a contribution in the backward case we propose caching more information to be able to compute any reachability heuristic without repeating the reachability analysis every time a state is evaluated. In particular, we cache the best supporters determined by Equation 3.1[2], which defines how to select the best supporter a_p of a proposition p among the set of actions $A(p)$ that add p (assume that $h^{max}(x, y)$ is h^{max} of $x \subseteq S$ computed from $y \subseteq S$).

$$a_p \in \operatorname*{argmin}_{a \in A(p)} \; (cost(a) + h^{max}(pre(a), I)) \tag{3.1}$$

The cost of any proposition $p' \in pre(a)$ is recursively defined as:

$$cost(p') = \begin{cases} 0 & \text{if } p' \subseteq s_0 \\ cost(a_{p'}) + h^{max}(pre(a_{p'}, I)) & \text{if } p' \nsubseteq s_0 \end{cases} \tag{3.2}$$

[2]Ties in Equation 3.1 are broken arbitrarily.

Caching the best supporters allows computing the FF heuristic (Hoffmann and Nebel, 2001), whose value is the cost of a plan that reaches a partial state from the source state in a delete-free version of the problem. Being able to compute the FF heuristic is important because h^{add} tends to overestimate the cost to the goal by a significant margin in some domains, which makes the FF heuristic have a better overall behavior (Keyder and Geffner, 2009b). Figure 3.5 shows the difference between caching costs in HSPr and best supporter caching. In this problem an agent must go from I to K to pick up a key and then carry it to G. h^{add} adds the distance from I to K to the distance from I to G and to the cost of picking up the key. This way, if the cost of all the actions is one, then $h^{add} = 9$. FF computes a relaxed plan by tracing back from the fluents of the partial state to the initial state using best supporters. Moving from I to K and G with no deletes requires 5 actions, so $h^{FF} = 6$. In either case, if the partial state changes, no new reachability analysis is needed; both heuristics can be computed with the cached information.

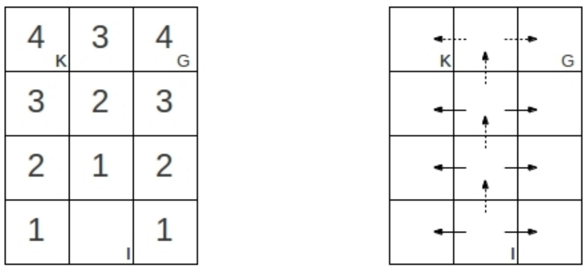


Figure 3.5: Caching costs (left) and best supporters (right); dashed arrows are part of the relaxed plan.

In terms of computational cost, the cached version of the FF heuristic has a complexity linear on the number of actions of the problem $|A|$, as opposed to the regular FF heuristic in progression, which has a complexity polynomial on $|A|$. Actually it is easy to see that FF in regression dominates FF in progression in terms of time, as the progression computation requires the reachability analysis and the plan retrieval, whereas the regression computation requires only the plan retrieval.

Preferred Operators in Regression

Being able to cache a broader range of information and to extract relaxed plans has other advantages too. In particular, structural information other than the mere numeric value derived from the computation of the heuristic may be interesting. We propose the use of one of the most significant techniques in terms of its potential to improve the efficiency of the search, *preferred operators* (Hoffmann and Nebel, 2001; Helmert, 2006; Richter and Helmert, 2009); note however that other similar techniques, like *look-aheads* (Vidal, 2004b), are equally possible to implement using caching schemes.

Preferred operators are an aggregated set of actions extracted from the heuristic computation. Originally they were proposed in the context of relaxed plans under the name of *helpful action* (Hoffmann and Nebel, 2001), although they are also computable for other reachability heuristics (Helmert, 2004; Keyder and Geffner, 2008; Helmert and Geffner, 2008). Preferred operators are a subset of the applicable actions of the state that allows discriminating between successors. Possible ways of exploiting them are: pruning successors generated with non-preferred operators (Hoffmann and Nebel, 2001); using multiple queues as open lists with a boosting scheme (Röger and Helmert, 2010) to prefer the queues whose states were generated with preferred operators; or weighting with a penalizing factor the heuristic value of states generated with non-preferred operators (Lipovetzky and Geffner, 2012).

In Fast Downwards's implementation of the FF heuristic the set of preferred operators is the subset of applicable actions that also appear in the relaxed plan.[3] Taking this into account, we redefine preferred operators in regression as the applicable actions *in regression* that appear in the relaxed plan. In the former example depicted in Figure 3.5 moving to G from the left is the only preferred operator.

Although the definition in regression is almost identical to the forward case, in practice there are some subtle differences that must be taken into account. Following the original relaxed plan graph definition of the FF heuristic, in progression all the actions that appear at the first action level are applicable. The relaxed plan must contain actions from this first level, so there must be at least some action that is helpful. In regression however there is no guarantee that the actions that support the goals are applicable, to the extent that in fact there may not be preferred operators obtainable from the heuristic computation. This is due to the fact that best supporters of the goals are *relevant* to the propositions they achieve by definition, but they may not be *consistent* with the whole set of goals. A way of circumventing this is forcing that at least one or all the best supporters of the goals must be applicable if possible, although this changes the value and the computational cost of the heuristic.

Another important fact regarding preferred operators in regression is that, as actions are applied farther away from the source of the reachability analysis, the information they represent is less reliable because the loss of information from the relaxation of the problem accumulates. Therefore, using preferred operators in regression means trusting actions selected in the reachability analysis after the problem has been thoroughly relaxed. This makes preferred operators in regression less reliable or informed than their forward counterparts *a priori*, although the impact of the difference between preferred operators in progression and in regression can only be evaluated empirically.

[3]The original implementation in FF is slightly different: all the applicable actions that achieve some proposition made true by the first time at the first level of the relaxed planning graph by some action of the relaxed plan are helpful *a.k.a.* preferred.

3.5.2 Computing Heuristics in P^m

Reachability heuristics are usually computed by ignoring the *delete* effects of actions. These effects can be partially included by using the alternative version of the problem P^m (Haslum, 2009). P^m is a redefinition of the planning task in which every fluent in P^m is a set of fluents of size m. A key conclusion of Haslum's paper (Haslum, 2009) - to the extent that it is the title of the publication - is that h^{max} in P^m is equivalent to h^m in P^1. An important observation though is that this is not exclusive to h^{max}: any other reachability heuristic can be computed in P^m. In fact, any general method based on a reachability analysis, like the computation of landmarks (Hoffmann et al., 2004), is also computable on P^m (Keyder et al., 2010).

Computing heuristics in P^m is exponential on m, so in practice it is not viable to use them in progression, as it would require an expensive reachability analysis per evaluated state. Nevertheless and as pointed out in this chapter, in regression only one reachability analysis is needed, which makes their computation tractable thanks to caching.

In order to generalize the reachability analysis, we propose a general definition of best supporters in P^m that takes into account sets of propositions of size m based on Equation 3.1. Best supporters are now determined by Equation 3.3, that defines the best supporter a_P of a set of propositions P with size $|P| \leq m$.

$$a_P \in \underset{a \in A(P)}{\text{argmin}} \; (cost(a) + h^m(Reg(P, a), I)) \tag{3.3}$$

$A(P)$ is the set of actions that generate at least a proposition in P without deleting any other:

$$A(P) = \{a \in \mathcal{A} \mid add(a) \cap P \neq \emptyset \wedge del(a) \cap P = \emptyset\} \tag{3.4}$$

$Reg(P, a)$ is the result of regressing P through a: $Reg(P, a) = P \setminus add(a) \cup pre(a)$. The cost of reaching the partial state s from the initial state I is defined as:

$$h(s, I) = \sum_{a \in \pi(s,I)} cost(a) \tag{3.5}$$

where

$$\pi(P, I) = \begin{cases} \emptyset & \text{if } P \subseteq I \\ \pi(a_P, P, I) & \text{if } P \not\subseteq I, |P| \leq m \\ \bigcup\limits_{P_m \in \binom{P}{2}} \pi(P_m, I) & \text{if } P \not\subseteq I, |P| > m \end{cases} \tag{3.6}$$

and

$$\pi(a, P, I) = \{a\} \cup \pi(Reg(P, a), I) \tag{3.7}$$

Caching best supporters in P^2 allows to compute the FF heuristic per state in P^2 efficiently - we call this the FF^2 heuristic. The complexity of the computation of the relaxed plan is in fact the same as in P^1, linear on $|A|$, so *a priori* FF^2 incurs in no overhead other than the initial reachability analysis in P^2. Nevertheless, a reachability analysis in P^2 is already required to compute h^2 to find mutexes of size two, which means that in practice best supporters for atom pairs can be cached when h^2 is computed with no overhead.

We compute the FF^2 heuristic per state as follows: for the partial state $s \subseteq S$ we compute all atom pairs $P_{ij} \mid p_i, p_j \in s$ and $p_i \neq p_j$. The best supporter $a_{P_{ij}}$ of each atom pair P_{ij} is added to the relaxed plan and the atom pairs obtained from $Reg(P_{ij}, a_{P_{ij}})$ that are not true in I are added as open preconditions. This is repeated until no open preconditions remain.

From a theoretical point of view, the FF^2 heuristic and all the heuristics that can be computed in P^2 are closely related to the family of heuristics that consider semi-relaxation via a set of conjunctions of propositions C (Keyder et al., 2012). The heuristics computed in P^2 can be considered a particular case of semi-relaxation in which C is formed by all the pairwise combinations of the propositions in S. Taking this into account, and since caching schemes can apply to any kind of reachability analysis, it may be a good idea to extend C with conjunctions of propositions of size bigger than two and cache the best supporters, although this is out of the scope of this section.

3.5.3 Reasonable Orders, Contexts and Paths through Invariants

Reasonable orders between goal propositions were first proposed to create a goal agenda (Koehler and Hoffmann, 2000) and later on extended to landmarks and arbitrary pairs of facts (Hoffmann et al., 2004). As pointed out in Section 2.5.4 no proper formal definition of reasonable orders is considered standard. Furthermore, several computation methods have been proposed to find reasonable orders. This makes difficult evaluating both the significance and the effectiveness of reasonable orders. As an attempt to standardize the definition of reasonable orders and to streamline their computation here we propose a more general definition that takes into account *e-deletion*. This covers both the formal aspect of reasonable orders and their implementation and allows us a simple and general way to extend the definition to sets of facts.

Definition 13. *(Reasonable Order) A proposition p is reasonably ordered before a set of propositions P_r ($p <_r P_r$) if all the supporters of p are e-deleters of some proposition $p' \in P_r$. Formally, $p <_r P_r$ if $\forall a \in A \mid p \in add(a)$: $e\text{-}del(a) \cap P_r \neq \emptyset$.*

Also it is easy to see that this novel definition of reasonable orders complies with the formerly proposed Definition 5: if $p <_r p'$ and p' is true in the current state, every path that achieves p must contain a supporter $a \in A$ of p - which in turn will be an *e-deleter* of p' - which means that after the execution of a it is guaranteed that p' will be false. To adapt this definition to landmarks or to any other case in which only the first time a pair of propositions is achieved is relevant - see Section 2.6 -, it suffices to consider only first achievers instead of the whole set of achievers as potential *e-deleters*.

Furthermore, all work on reasonable orders was done with progression in mind, so the implications that reasonable orders may have in regression have not been explored. Reasonable orders were deemed "unsound" in progression, because it is not true that, if $p <_r p'$, then p must be achieved first in every solution plan. Nevertheless, some sound information can be derived: if $p <_r p'$ and both fluents must be true until the end of the plan, then p' must be achieved *last*. This is closely related with the concept of *aftermatch* (Koehler and Hoffmann, 2000), that is, whether the value of the involved propositions changes or must be kept untouched until the end of the plan. In progression this can only be proved for top level goals. However, in regression this holds for every $p <_r p'$ if p and p' are true in a partial state, as partial states in regression can be treated as top level goals. The main implication is that, as p' must be achieved last, then in regression p' must be supported first, which means that *reasonable orders are actually sound in regression.* A straightforward way of exploiting reasonable orders in regression is that, if $p <_r p'$ and p, p' are true in a partial state, then the supporters of p can be safely pruned.

Theorem 3. *$\forall p, p' \in S$, $p <_r p'$ and $p, p' \in s \subseteq S$, then $\forall a \in A \mid p \in add(a) : a$ can be safely pruned.*

Proof. If $p <_r p'$ then $\forall a \in A \mid p \in \text{add}(a) : p' \in \text{e-del}(a)$ following Definition 13. This way, $\forall a \in A \mid p \in \text{add}(a) : a$ is not consistent and consequently not applicable in regression in s following Definition 12. $\square$

Although this property of reasonable orders may look strong, if consistency as defined in Definition 12 is used during search then no additional pruning is obtained. Nevertheless, the soundness of reasonable orders in regression can be exploited in other ways, like in Section 7.3.1 or as described next.

When computing heuristics, reasonable orders in regression allow us inferring stronger precedence constraints. The context-enhanced additive heuristic, h^{cea} (Helmert and Geffner, 2008), is a generalization of h^{add} and the causal graph heuristic h^{cg} (Helmert, 2004) that considers additional information to get more accurate values. In h^{cea} several contexts or pivots are selected during the heuristic computation and the cost of some goals is computed from a state that results from achieving the pivots instead of from the evaluated state. Let's see a small example: if we go back to Figure 3.5 we can see the most informative context is *agent-at=K*. This is so because the steps that must be taken to achieve *agent-at=G* must be counted from *K* instead of from *I*, since the agent must go first to *K* to pick up the key. In fact, if *agent-at=K* is chosen as context for h^{cea} then h^{cea} would add the distance from I to K to the distance from K to G and to the cost of picking up the key, yielding $h^{add} = 7$, the cost of the optimal solution.

Choosing the right context is still an open question though. Reasonable orders were already proposed as a way of computing contexts (Cai et al., 2009), although in progression they may be misleading because they do not have to be respected in every solution plan. In regression though, if $p <_r p'$ and p and p' are true at the same time, p' must be supported after having achieved p. This hints from which context p' is achieved in most solution plans: p. This assumption does not shed light on how to implement such a precedence constraint, so we propose a more systematic way: a likely context in the computation of h^{cea} for a partial state $s \subseteq S$ may be the conflicting propositions that cause the *e-deletion* of any $p' \in s$ by the best supporter of some other $p \in s$ (Nguyen and Kambhampati, 2001). Going back again to the previous example in Figure 3.5, the best supporter of *have-key* is *pick-key(K)*. *pick-key(K)* has as precondition *agent-at=K*, which is the cause of the *e-deletion* of *agent-at=G* because *agent-at=K* and *agent-at=G* are mutex. This suggests that *agent-at=K* is indeed a useful context, even more so considering that in a multi-valued representation *agent-at=K* and *agent-at=G* will most likely belong to the same variable $v \in \mathcal{V}$.

In order to exploit the precedence constraints derived from reasonable orders in regression, an efficient computation of h^{cea} is also needed. Anyway, best supporter caching is also possible in h^{cea} by caching supporters when traversing Domain Transition Graphs (DTGs) in a similar way as h^{cg} does with costs. The procedure is as follows: assume that $p \in s \subseteq S$ is a proposition true in the evaluated state s, $p' \in S$ is a proposition conflicting with p and both propositions belong to the same invariant group $p, p' \in \theta$. Then, uniform-cost search is performed in the relaxed DTG defined by θ with p' as the source until the optimal distance from p' to every other proposition in θ is obtained. When uniform-cost search is performed, whenever a proposition $p'' \in \theta$ is expanded we cache the last operator of the path that goes from p' to p'' as the best supporter.

The cached version of h^{cea} for a state $s \subseteq S$ is thus computed as follows:

- Get all the pairs of conflicting propositions $\{p, p'\} \mid p \in s$ and p is mutex with p' by checking which propositions of s are *e-deleted* by the best supporter of another proposition of s.
- For each conflicting pair in the same invariant group $\{p, p'\} \mid p, p' \in \theta$, if uniform-cost search from p' in the DTG derived from θ has not been performed for some previous heuristic evaluation, perform it.
- For each conflicting pair $\{p, p'\}$, retrieve the relaxed plan from p' to p.
- Once this is done, compute the cached h^{FF} for a state $s' \subseteq S$ obtained by substituting in s every proposition $p \in s$ that has a conflict with the pivot p' obtained from the conflict pair $\{p, p'\}$
- Return the union of the relaxed plan from the h^{FF} computation with every relaxed plan obtained from a conflict pair $\{p, p'\}$

If cached h^{cea} is computed in the example shown in Figure 3.5 the procedure would follow the subsequent steps:

- The conflicting pair $\{$*(agent-at=K),(agent-at=G)*$\}$ is found by looking at the best supporter of *(have-key)*, which is *pick-key(K)*. *pick-key(K) e-deletes (agent-at=G)* because it has *(agent-at=K)* as precondition and *(agent-at=K)* and *(agent-at=G)* are mutex.

- Uniform-cost search in the DTG obtained from the invariant group that corresponds to *(agent-at=K)* and *(agent-at=G)* is done to cache the best supporters from *(agent-at=K)*.
- The relaxed plan RP_{DTG} from *(agent-at=K)* to *(agent-at=G)* is retrieved.
- The alternative state that substitutes the goals by their contexts is computed: $\{$*(agent-at=K),(have-key)*$\}$ substitutes $\{$*(agent-at=G),(have-key)*$\}$.
- The relaxed plan RP_{FF} for $\{$*(agent-at=K),(have-key)*$\}$ from I is computed.
- Both relaxed plans are aggregated $RP_{DTG} \cup RP_{FF}$ and returned.

As stated before h^{cea} returns in this problem the optimal cost - in fact, the relaxed plan is an optimal solution plan of the problem. Note that this implementation of h^{cea} is not a faithful rendition of its version in progression: in this case the costs derived from the reachability analysis are not added but rather aggregated as done for h^{FF}. This is done for two reasons: first, to avoid the overestimation of costs caused by additive schemes; and second, to allow extracting preferred operators in a more straightforward way.

3.6 Experimentation

The proposed techniques were implemented on top of Fast Downward into a new planner called FDr (Fast Downward Regression). We used the benchmark suite from IPC2011 and compared FDr against HSPr, Fast Downward (FD) with greedy best-first search (GBFS), h^{FF} and delayed evaluation and Mp (Rintanen, 2010). Mp was included because it includes several modifications that make the SAT solver prefer actions that support open goals and preconditions not unlike what search algorithms in regression do. Four configurations were tested: FDr^{FF}, FDr^{add}, FDr^{cea}, FDr^{2}, which use h^{FF}, h^{add}, h^{cea} and h^{FF} in P^{2} respectively. FDr's search algorithm is GBFS with regular evaluation. Since the focus of the experimentation is coverage, action costs were ignored. Results are shown in Table 3.1.

Using disambiguation to prune spurious operators reduces the number of actions in six domains: *tidybot*, *woodworking*, *nomystery*, *scanalyzer*, *barman* and *parking*. The number of spurious operators pruned for these domains is not negligible: the geometric mean of the percentage of pruned operators is around 85% in *nomystery* and between 50% and 10% for the rest of the domains. Additionally, in *nomystery* a few extra mutexes are found if h^2 is recomputed without the pruned operators and in *tidybot* some fluents are found to be unreachable.

Preferred operators were not used. Although promising *a priori*, they were not helpful. This may seem counterintuitive, as the reduced branching factor and the additional heuristic guidance should help, but in partial states they often commit strongly to unpromising areas of the search space. This is due to the problems

Domain	FD	FDr$^{\text{FF}}$	FDr$^{\text{add}}$	FDr$^{\text{cea}}$	FDr^2	HSPr	Mp
barman	18	0	0	0	0	0	6
elevators	18	16	16	10	0	**20**	17
floortile	3	**20**	**20**	**20**	**20**	18	**20**
nomystery	9	6	6	7	5	4	17
openstacks	**20**	0	0	0	1	0	0
parcprinter	11	12	12	12	**20**	11	**20**
parking	19	5	0	4	0	3	0
pegsol	**20**	14	11	15	11	10	**20**
scanalyzer	17	**20**	**20**	17	19	**20**	17
sokoban	19	3	2	3	2	3	2
tidybot	14	1	0	0	0	0	17
transport	0	5	0	2	0	0	0
visitall	4	5	**17**	3	7	5	0
woodworking	19	19	19	11	16	17	**20**
Total	**191**	126	128	104	101	111	156

Table 3.1: Coverage in the IPC11 benchmark.

already presented in Section 3.5.1. First, actions at the later levels of the reachability analysis suffer from the accumulated relaxation of the problem and so are less useful to guide the search. Second, there is no way of knowing how many goal propositions will be supported by applicable actions, which means that the number of preferred operators may vary and there may be states for which there are no computable preferred operators. When enabled, FDr^{FF} only solved one more problem in *woodworking* while losing coverage in *parcprinter*, *transport* and *visitall*. This is also the reason why for a fairer comparison FD in progression does not use preferred operators either.

As seen in Table 3.1, FDr^{FF} solves consistently more problems than HSPr and is not far from Mp in coverage. FDr^2 performed worse except on *parcprinter*, in which it solved the whole set of problems, and *floortile*. The overhead of h^{cea} does not pay off, surpassing other heuristics only in two domains. No regression planner was able to solve any instance in *barman* due to spurious states that could not be detected (*i.e.* due to some imprecisions in the domain formulation h^2 cannot detect mutexes like a shaker being clean and containing an ingredient at the same time). The regression planners fare worse than FD in total coverage, although this varies among domains. FDr is superior to FD in *floortile*, as this domain contains a high number of dead ends in progression that are difficult to detect. FDr^{add} also solves many more problems in *visitall*, a domain in which heuristics like FF are affected by big plateaus. *openstacks* and *sokoban* are the opposite case: problems trivial for FD are hard for all the regression planners. As hypothesized, Mp seems to behave similarly to regression planners, with the exceptions of *nomystery* and *tidybot*.

An ablation study was also done disabling disambiguation and decision trees as successor generators. Eight fewer problems were solved using FDr^{add} after disabling disambiguation. The number of expansions increased by an order of magnitude in *parcprinter* and *woodworking*, staying the same or very similar in other domains. When disabling decision trees, five fewer problems were solved. The impact was proportional to the number of grounded actions and the number of effects of those actions, increasing search time by an order of magnitude in domains like *floortile* and *woodworking*.

The IPC11 domains were criticized for being very sequential domains, which hurts the parallel SAT encoding of Mp. Therefore, another informal experiment was done in the STRIPS domains where Mp reportedly was considerably better than Fast Downward (Rintanen, 2010): *airport*, *storage* and *trucks*. Table 3.2 shows the comparison between LAMA 2011 (Richter and Westphal, 2010), Fast Downward with the FF heuristic, delayed evaluation and preferred operators (FD), Mp, FDr and FDr^2. Note that both LAMA and FD used preferred operators, which usually have a noticeable positive impact on the search. Mp is the clear winner, but FDr and FDr^2 are still competitive with LAMA and Fast Downward in the three domains. Again, this suggests that FDr is somewhere between Mp and Fast Downward regarding how it behaves.

Domain	Mp	FDr	FDr^2	LAMA	FD
airport(50)	50	32	26	31	34
storage(30)	30	25	20	19	20
trucks(30)	21	16	9	15	19
Total	91	73	45	65	73

Table 3.2: Coverage in selected domains. Number of instances in parenthesis.

3.7 Related Work

Although most planning benchmarks are not symmetrical, it is possible to reverse a planning problem by following the method proposed by Massey (1999) and employed by Pettersson to build a reversed version of the planning graph (2005). Massey's method builds a reversed version $R(P)$ of the original *STRIPS* P in polynomial time and space such that any solution plan $(a_1, a_2, \dots, a_n)$ in P is also a valid plan in $R(P)$ and *vice versa.* The key aspect of this formulation is that for every $p \in S$ it allows p and $\neg p$ to be true at the same time, and whenever the value of p becomes unknown (for example, when it is added by an action but it does not appear in the preconditions), both p and $\neg p$ are made true. $R(P)$ can be solved by any progression planner, although it shares the same drawbacks as doing regression in P: spurious states, subsumption of partial states and less informative heuristics. Besides, it does not exploit the advantages that regression has, like the caching of the reachability analysis. For instance, if FD was used to solve $R(P)$ it would have less accurate heuristics (because the source of reachability analysis would be a partial state in which p and $\neg p$ would be true for unknown propositions), it would generate preferred operators that would lead to spurious states (in fact there might be cases in which all the preferred operators lead to spurious states), it would recompute the reachability analysis every time a state is evaluated, and so on. Therefore, for a proper analysis of the regression case it is advisable not to employ $R(P)$ but rather adapt (or develop) specific techniques for regression.

Curiously enough, in the context of $R(P)$, in particular when computing h^2 backwards, Haslum's HSPf (Haslum, 2008) uses mutexes computed with h^2 forward to reduce the cases in which p and $\neg p$ are both true, and even proposes the use of invariant groups as an improvement. This is the same as disambiguation, although with a less ambitious scope: it was proposed just to complement the set of goal propositions instead of any arbitrary set of propositions in order to obtain a more complete planning model, and did not inquire about the cases in which disambiguation allows detecting spurious sets of actions.

Caching schemes for the reachability analysis are not a completely novel concept either. GRT (Refanidis and Vlahavas, 2004) is a progression planner that computes a variant of h^{add} with a backward reachability analysis. As the source of the reachability analysis is the set of goal propositions and these do not change during search then the costs are cached to avoid repeating the heaviest part of the computation of the heuristic. The main drawback is that, as pointed out before, a reachability analysis from the goals is much less accurate due to incomplete states and proper consistency checks. The planner AltAlt (Nigenda et al., 2000) proposes a caching scheme similar to FDr's, in which the planning graph is kept in memory to compute a more complex version of h^{add} that takes into account negative effects and to extract actions likely to guide the search like preferred operators do.

3.8 Conclusions

In this chapter we analyzed several state-of-the-art techniques in forward search and explored their potential in regression. Some novel definitions were also proposed. We implemented a new regression planner, FDr, which performed consistently better than its predecessor HSPr. Forward search planners still have the upper hand in most domains, although their behavior varies among domains. Overall performance seems to depend greatly on the topology of the search space, which means that one should not rule out regression in domain-independent planning. This also implies that there may be a high synergy between FD and FDr, potentially leading to a bidirectional planner better than the combination of the individual planners in a portfolio. The implementation of such a planner remains as future work along with the use of other state-of-the-art techniques which are simple to employ in regression, like landmarks (Richter and Westphal, 2010) and multiple queues (Röger and Helmert, 2010).

Chapter 4

State Invariants in Symbolic Search

Symbolic search allows saving large amounts of memory compared to regular explicit-state search algorithms. This is crucial in optimal settings, in which common search algorithms often exhaust the available memory. So far, the most successful uses of symbolic search have been bidirectional blind search and the generation of abstraction heuristics like Pattern Databases. Despite its usefulness, several common techniques in explicit-state search have not been employed in symbolic search. In particular, mutexes and other constraining invariants, techniques that have been proven essential when doing regression, are yet to be exploited in conjunction with Binary Decision Diagrams (BDDs). In this chapter we analyze the use of such constraints in symbolic search and its combination with minimization techniques common in BDD manipulation. Experimental results show a significant increase in performance, considerably above the current state of the art in optimal planning.

4.1 Introduction

In optimal planning, in which a plan of least cost must be found, the most popular approach is using A* (Hart et al., 1968) combined with an admissible heuristic. The main shortcoming of A* is its memory requirements. When using heuristics in optimal search, all the nodes whose *f-value* is less than the cost of the optimal solution must be expanded (unless they are pruned by an optimality-preserving pruning technique). Hence, if the heuristic is not accurate enough, the number of generated nodes may exceed what the main memory can store.

Several alternatives to A* have been proposed, like symbolic search, which uses Binary Decision Diagrams (BDDs) (Bryant, 1986) to represent sets of states instead of storing them individually. When using BDDs, a potentially exponential saving in memory may be obtained. This means that symbolic versions of common search algorithms, like symbolic blind search and BDD-A* (Edelkamp and Reffel, 1998), are often able to solve problems that the explicit-state versions are unable to solve due to memory problems.

The main uses of symbolic search have been so far regular bidirectional search and the generation of abstraction heuristics (Edelkamp and Reffel, 1998; Torralba et al., 2013b) for their use in both symbolic and explicit search algorithms. These methods require performing regression on the goals of the problem. Regression in planning is known to be less robust than progression due to the existence of multiple goal states, the presence of partially-defined states and the impact that spurious states have on the search (Bonet and Geffner, 2001). To alleviate this, constraints obtained from state invariants of the problem such as binary static mutexes and *"exactly-1"* invariant groups have been thoroughly employed (Bonet and Geffner, 2001; Haslum et al., 2007). Surprisingly enough, the use of these constraints has not been extrapolated to symbolic search except for the monotonicity analysis required to transform the planning task into SAS$^+$ (Kissmann and Edelkamp, 2011). Although the size of a BDD does not have to be proportional to the number of states it contains, there may exist a correlation, in which case constraints may help.

Also, the use of constraints is commonplace in BDD manipulation. Some minimizing operations have been specifically designed to reduce the size of a BDD when subject to a given constraint in the form of another BDD. The use of these minimization operations may translate not only into smaller (and thus more memory-efficient) BDDs but also in faster BDD manipulation, as the time consumed by the logical operations performed over BDDs depends on their size. Taking all this into account, in this work we study the impact of constraints such as mutexes and invariant groups in symbolic search and symbolic abstractions and how minimization of BDDs affects the performance of these techniques.

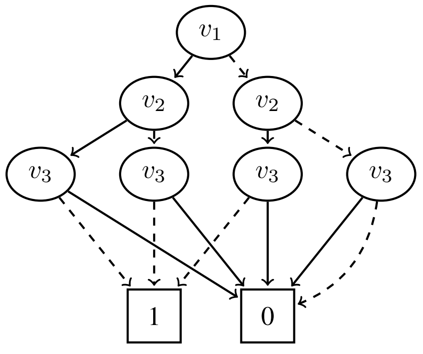
4.2 Symbolic Search and Binary Decision Diagrams

In symbolic search, originally proposed for model checking (McMillan, 1993), operations are performed over sets of states instead of working with individual states. The main motivation is taking advantage of succinct data structures to represent sets of states through their characteristic functions. The characteristic function returns true if and only if the state belongs to the set of states. The use of a characteristic function also allows the planners to operate with set of states through function transformations. For example, the union ($\cup$) and intersection ($\cap$) of sets of states are derived from the disjunction ($\vee$) and conjunction ($\wedge$) of their characteristic functions, respectively. Also, the complement set (X^c) corresponds with the negation ($\neg$) of the characteristic function. For simplicity, in this chapter we will make no distinction between a set of states and its characteristic function.

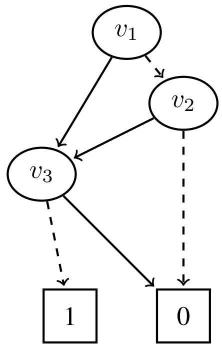
Decision Diagrams are popular data structures for symbolic search inspired by the graphical representation of a logical function. A Binary Decision Diagram

(BDD) is a directed acyclic graph with two terminal nodes (called *sinks*) labeled with *true* and *false*. All the internal nodes are labeled with a binary variable $v \in \mathcal{V}$ and have two outgoing edges that correspond to the cases in which v is *true* and *false* respectively. When representing a multi-valued variable $v \in \mathcal{V}$, $\lceil \log_2 |D_v| \rceil$ binary variables are used instead. For any assignment of the variables on a path from the root to a sink, the represented function will be evaluated to the value labeling the sink. A more general type of decision diagram is an Algebraic Decision Diagram (ADD) (Bahar et al., 1997), which can have an arbitrary number of different sink nodes. This allows evaluating the represented function to values different from *true* and *false*.

Variables in a BDD are ordered. This has three advantages: first, the operations performed between BDDs with the same variable ordering are quadratic in the worst case; second, for a given set of states, an ordered BDD guarantees uniqueness; third, it allows a more succinct representation of the set thanks to reducing operations. The reducing operations of BDDs are the *deletion rule*, in which nodes with outgoing edges that lead to the same successor are removed, and the *merging rule*, in which nodes labeled with the same variable are merged if their respective successors are the same. Figure 4.1 shows how the size of a BDD can be decreased using reducing operators. In this case, all the nodes labeled with v_3 except for the one down the path $\neg v_1 \wedge \neg v_2$ have the same successors, so they are merged. After that, both the node labeled with v_2 down the path v_1 and the node labeled with v_3 down the path $\neg v_1 \wedge \neg v_2$ have outgoing edges that lead to the same successor, so they are removed.



(a) Original BDD



(b) Reduced BDD

Figure 4.1: Example of BDD reduction rules.

The size of a BDD is fully determined by the variable ordering. There can be an exponential difference in size between two different variable orderings for the same function (Bryant, 1986, 1992). However, finding the variable ordering that minimizes the size of a BDD is *co-NP-complete* (Bryant, 1986). Also, the variable ordering is fixed throughout the search, which means that a good ordering for the first layers may be inadequate for later layers. In this work we use the Gamer's strategy to select a variable ordering (Kissmann and Edelkamp, 2011) in a preprocessing stage, which is empirically among the best static orderings for planning (Kissmann and Hoffmann, 2013).

The set of operators $\mathcal{O}$ is represented using one or more *Transition Relations* (TRs). A TR is a function defined over two sets of variables, one set x representing the *from*-set and another set x' representing the *target*-set. Any given TR represents one or more operators $o \in \mathcal{O}$ with the same cost. To compute the successors of a set of states S_g, the *image* operation is used. The definition of *image* is as follows:

$$\text{image}(S_g, TR_i) = \exists x \ . \ S_g(x) \wedge TR_i(x, x')[x' \leftrightarrow x]$$

Thus, *image* is carried out in three steps:

1. The conjunction with $TR_i(x, x')$ filters preconditions on x and applies effects on x'.
2. The existential quantification of the predecessor variables x removes their values relative to the predecessor states.
3. $[x' \leftrightarrow x]$ denotes the swap of the two sets of variables, setting the value of the successor states in x.

Similarly, regression uses the *pre-image* operator:

$$\text{pre-image}(S_g, TR_i) = \exists x' \ . \ S_g(x') \wedge TR_i(x, x')[x \leftrightarrow x']$$

4.2.1 Encoding State Invariants as BDDs

Ever since the first application of heuristic backward search in domain-independent planning, pruning spurious states has been considered essential (Bonet and Geffner, 2001). Binary mutexes allow pruning spurious states that otherwise would be considered for expansion during search. Expanding such states may lead to an exponential decrease in performance, as none of the successors of a spurious state may lead to the initial state by doing regression. The use of mutexes in explicit-state search is straightforward: simply prune every state s such that fluents $f_i, f_j \in s$ are mutex. The use of invariant groups requires disambiguating the state as previously described.

Despite the impact that the use of state invariant has in explicit-state regression, this technique has not been employed in symbolic search. Although it is obvious that a *per state* application of mutexes in symbolic search is impossible, there are alternatives. In particular, we propose *creating a BDD that represents in a succinct way all the states that would be pruned if state invariants were used.* This BDD, that we call the *constraint BDD* ($cBDD$), can be used to discard all the states that have been generated using a TR in a similar way as it is done with the closed list for duplicate detection purposes.

$cBDD$ is created as follows: every binary mutex is a conjunction of fluents $f_i, f_j | f_i \neq f_j$ such that, if $f_i, f_j \in s$ in state s, then s is spurious. Hence, the set of states that can be pruned using mutexes consists of those in which at least one such conjunction of fluents is true. This way, the logical expression represented by $cBDD$ is the disjunction of all the mutexes found with h^2 (represented by the conjunction of both fluents).

Constraints derived from invariant groups are encoded in a similar way. Given an invariant group $\theta = f_1, \dots, f_k$, two types of constraints may be deduced: first, the set of all the mutexes of the form $f_i \wedge f_j$ if $f_i \neq f_j$; second, the fact that at least one fluent $f_i \in \theta$ must be true in every non-spurious state. The first constraint overlaps with the mutexes computed with h^2, so it does not make sense to include it in the $cBDD$. The latter however can be encoded as an additional constraint of the form $\neg(f_1 \vee f_2 \vee \dots \vee f_k)$ for every invariant group. Such a constraint can be included in the *constraint BDD*, allowing to prune spurious states like the ones pruned by disambiguation.

Formally, if M is the set of binary mutexes found by h^2 and I_g the set of *at-least-one* constraints, then:

$$cBDD = \left(\bigvee_{<f_i, f_j> \in M} f_i \wedge f_j \right) \vee \left(\bigvee_{<f_1, \dots, f_k> \in I_g} \neg f_1 \wedge \dots \wedge \neg f_k \right)$$

Even though each individual constraint is efficiently representable, the size of $cBDD$ is exponential on the number of encoded constraints in the worst case. This follows from the results provided by Edelkamp and Kissman (2011) on the complexity of the representation of partial states in BDDs. To ensure that we can represent $cBDD$, we set a limit on the number of nodes $cBDD$ can contain and, if that limit is surpassed, we divide $cBDD$ into k BDDs: $cBDD = cBDD_1 \vee cBDD_2 \vee \dots \vee cBDD_k$ such that every $cBDD_k$ is below the given limit.

The merging process is initialized with the BDDs that represent individual constraints. These BDDs are aggregated until they are larger than the given threshold. The order in which these disjunctions are applied affects the efficiency of the procedure and the number of BDDs used to represent $cBDD$. To try to obtain BDDs as small as possible, for each variable $v_i \in \mathcal{V}$ we compute a BDD describing all binary mutexes of fluents relative to both v_i and v_j with $j > i$. Then, we iteratively merge the BDDs of mutex constraints related to each variable.

Spurious states determined by state invariants are pruned by computing the difference $S_g \setminus cBDD$ of a newly generated set of states S_g with the constraint BDD $cBDD$. In terms of BDD manipulation this is done by computing the logical conjunction of S_g with the negation of $cBDD$: $S_g \wedge \neg cBDD$. This operation is the same as the one done in symbolic search with the BDD that represents the set of closed states, used to prune duplicates. Extending the operation to the case where we have more than one $cBDD$ is straightforward: $S_g \wedge \neg cBDD_1 \wedge \dots \wedge \neg cBDD_k$.

In the rest of the chapter we assume that $cBDD$ is represented in a single BDD, without loss of generality.

An important remark about the usefulness of pruning states with $cBDD$ is necessary, though. A priori, there is no correlation between the number of states that a BDD represents and the size of the BDD. This means that there is no guarantee that pruning spurious states will help in symbolic search, as opposed to the explicit-state case. However, the representation of a state invariant in $cBDD$ is in most cases very succint and can be used to prune an exponential number of states. Figure 4.2a exemplifies this case; to encode a binary mutex of the form $\neg(v_1 \wedge v_2)$ only a BDD with two inner nodes is needed. A single state usually requires more nodes, so if that state is spurious because it violates $\neg(v_1 \wedge v_2)$ then using $cBDD$ is more compact than representing the state itself.

Of course, when multiple states are involved estimating the increase in compactness obtained by using $cBDD$ is more complex. Still, it is plausible to assume that *a priori* a similar phenomenon will occur. Let's assume that a BDD S_g, shown in Figure 4.2b, represents a set of states that contains spurious states that violate $\neg(v_1 \wedge v_2)$. If that is the case, every state represented by any path that satisfies $v_1 \wedge v_2$ could be safely pruned. In practice, this means that the subgraph whose root is the node that corresponds to v_3 at end of that path could be "removed", with a significant potential to reduce the size of S_g. If we look at Figure 4.2c, we can see that computing the difference $S_g \setminus cBDD$ has this exact effect, as evaluating v_2 after following the path that satisfies $v_1 \wedge v_2$ leads directly to the *false* sink node.

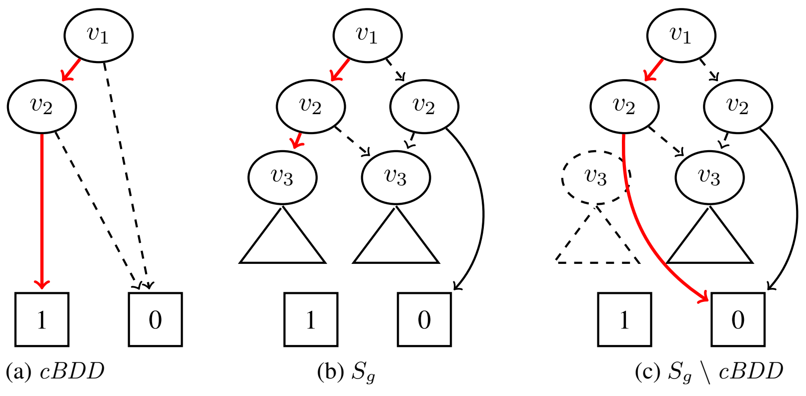
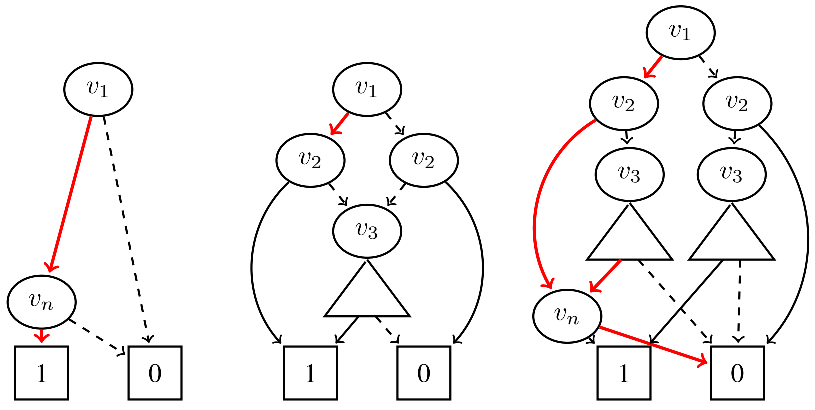


Figure 4.2: Positive case when using $cBDD$.

Again, there is no guarantee that the resulting $S_g \setminus cBDD$ will be smaller: first, computing $S_g \setminus cBDD$ may increase the size of the BDD if the added constraints make reference to variables in $\mathcal{V}$ that did not appear initially in S_g; second, as the "removed" subgraph is not necessarily isolated – in the sense that it may overlap with other subgraphs of S_g – the gain in compactness obtained from merging nodes of the “removed” subgraph with other nodes of S_g may be lost.

Another factor that might have negative impact is the variable ordering, when the variables of $cBDD$ are not close together, like in Figure 4.3a. In this case the subgraph in S'_g whose root corresponds to v_3 must be split in two to represent that it actually matters whether it is reached by a path containing v_1 or $\neg v_1$, as there is a constraint relevant to v_1.



(a) $cBDD$ (b) S_g (c) $S_g \setminus cBDD$

Figure 4.3: Negative case when using $cBDD$.

Of course, when multiple constraints are involved, estimating the increase in compactness obtained by using $cBDD$ is more complex. Alternatives to the computation of the difference $S_g \setminus cBDD$ will be analyzed in Section 4.3.

4.2.2 Pruning Spurious Operators in Symbolic Search

The use of state invariants as a constraining technique is not limited to states generated in regression. As shown in Section 3.4.1, any partial state can be detected as unreachable by disambiguating it, including action preconditions. An alternative to solving a CSP to prune spurious operators exists in symbolic settings. The symbolic representation of partial states does not consider undefined values, so disambiguation is automatically resolved when removing spurious states from any set of states (using $cBDD$, for example). Therefore, spurious operators may also be found by encoding all the constraints as prevail conditions in the TRs. An operator $o \in \mathcal{O}$ is detected as spurious if the conjunction $TR_o \wedge cBDD \wedge cBDD[x \leftrightarrow x']$ is empty.

4.2.3 Encoding Constraints in the TRs

State invariants allow pruning spurious states after they are generated. However, more efficient alternatives that avoid the generation of spurious states exist. For instance, the use of *e-deletion*, another invariant of the problem, allows avoiding the generation of spurious states in explicit-state search by modifying the definition of applicability in regression, as proposed in Section 3.4.2. In this section we study how to encode constraints in the TRs so that states that violate some constraint are never generated, in the spirit of *e-deletion*.

First, we identify which constraints must be encoded in each operator $o \in \mathcal{O}$. This comes from the fact that, although replicating all the constraints in the TRs is possible, this may lead to a great degree of redundancy. We show that, by assuming that the set of states to be expanded does not contain spurious states, we can safely consider only a subset of constraints in each TR while avoiding the generation of spurious states. Therefore, we denote a constraint as relevant for an operator if it may become violated after the application of the operator.

Definition 14. *(Relevant Constraint) A constraint c is relevant for an operator o if there exists a state s such that s satisfies c $(s \models c)$ and that the state that results from applying o in s in regression $(o(s))$ does not satisfy c $(o(s) \not\models c)$.*

This definition of relevant constraints allows us to define the *constrained* version of an operator. To avoid the generation of spurious states, the relevant constraints must be encoded as negative preconditions. This ensures that the encoded constraints are not violated after the *pre-image* computation, so that no state that violates the constraints is generated during the symbolic exploration.

Definition 15. *(Constrained Operator) Let $o \in \mathcal{O}$ be an operator and let C_o be the sets of constraints relevant with respect to o respectively.*

Then the constrained operator o^c derived from o becomes o with extra preconditions $pre(o^c) = pre(o) \wedge \bigwedge_{c_i \in C_o} c_i$.

Using o^c $\forall o \in \mathcal{O}$ suffices to guarantee that the successor set does not contain spurious states as long as the source set of states does not contain spurious states.

Theorem 4. *Let $S \subseteq \neg cBDD$ be a state set that does not contain states detected as spurious and o^c the constrained version of an operator $o \in \mathcal{O}$. Let S' be the resulting state set from applying o^c in regression to S. Then $S' \subseteq \neg cBDD$, that is, does not contain states that can be detected as spurious.*

Proof. Proof by contradiction. Suppose there is a state $s' \in S'$ that violates some constraint c. As c was satisfied by every state in S, by definition c is a relevant constraint for o. Then $c \in pre(o^c)$, so o is not applicable on s'. □

Propositions 1 and 2 show the sufficient and necessary conditions for mutexes and invariant groups to be relevant. Essentially the relevant constraints are those that contain fluents that may be *added* by o.

Proposition 1. *(Relevant mutex) Let M_i be a mutex of size m, $M_i = \neg(f_1 \wedge f_2 \wedge \cdots \wedge f_m)$. Let $o \in \mathcal{O}$ be a non-spurious operator and $\mathcal{V} \supseteq \mathcal{V}_u(o) = \{v_i \mid post(o)[v_i] \neq \mathbf{u} \wedge pre(o)[v_i] = \mathbf{u}\}$.*

Then M_i is relevant for operator o if for some fluent $f_i = \langle v_i, x\rangle \in M_i$, either (a) $f_i \in \mathit{pre}(o)$ or (b) $v_i \in \mathcal{V}_u(o) \wedge \mathit{post}(o)[v_i] \neq x$.

Proof. Suppose that M_i is relevant for o, that is, it is satisfied in $s' = o(s)$. Suppose that M_i is not satisfied in s. Then, for some fluent $f_i = \langle v_i, x\rangle$, $f_i \notin s'$, $f_i \in s$. Therefore, $s[v_i] \neq s'[v_i]$, which implies $\mathit{post}(o)[v_i] \neq \mathbf{u}$. Two cases are possible with respect to the preconditions of o:

(a) $\mathit{pre}(o)[v_i] \neq \mathbf{u}$. As o must be applicable in s, $\mathit{pre}(o)[v_i] = x$, so that $f_i \in \mathit{pre}(o)$.

(b) $\mathit{pre}(o)[v_i] = \mathbf{u}$ and $v_i \in \mathcal{V}_u(o)$ immediately follows.

□

Proposition 2. *(Relevant* at least one *invariant) Let m_{inv} be an* at least one *invariant, $m_{inv} = f_1 \vee f_2 \vee \cdots \vee f_k$.*

Let $o \in \mathcal{O}$ be an operator and $\mathcal{V} \supseteq \mathcal{V}_u(o) = \{v_i \mid \mathit{post}(o)[v_i] \neq \mathbf{u} \wedge \mathit{pre}(o)[v_i] = \mathbf{u}\}$.

Then m_{inv} is relevant for o if for some fluent $f_i = \langle v_i, x\rangle \in m_{inv}$, $f_i \in \mathit{post}(o)$ and (a) $\exists f' = \langle v_i, y\rangle \in \mathit{pre}(o), x \neq y$ or (b) $v_i \in \mathcal{V}_u(o) \wedge \mathit{post}(o)[v_i] \neq x$.

Proof. Suppose that m_{inv} is relevant for o in regression, i.e. is satisfied in $s' = o(s)$ but not in s. Then, for some fluent $f_i = \langle v_i, x\rangle \in m_{inv}$, $f_i \notin s$, $f_i \in s'$. Therefore, $s[v_i] \neq s'[v_i] = x$, which implies $\mathit{post}(o)[v_i] = x \neq \mathbf{u}$. Two cases are possible with respect to the preconditions of o:

(a) $\mathit{pre}(o)[v_i] \neq \mathbf{u}$. Then $\mathit{pre}(o)[v_i] = s[v_i] \neq x$, so that $\exists f' = \langle v_i, y\rangle \in \mathit{pre}(o), x \neq y$.

(b) $\mathit{pre}(o)[v_i] = \mathbf{u}$ and $v_i \in \mathcal{V}_u(o)$ immediately follows.

□

As aforementioned, for this property to hold we must ensure that the parent BDD does not contain spurious states. In progression this is always the case, as s_0 represents a single non-spurious state. On the other hand, $s_\star$ may contain spurious states, as $s_\star$ is in most cases partially defined. In this case, to guarantee that $s_\star$ does not contain spurious states, we compute the difference $s_\star \setminus \mathit{cBDD}$ to remove them from the goal description. When constraints are encoded in the TRs, this difference must be computed only once, before starting the search. This means that there will be no further overhead because of the use of cBDD, with the additional advantage that cBDD can be discarded afterwards to free memory.

4.3 BDD Minimization

The main motivation for using a *constraint BDD* is to remove spurious states so the BDDs that represent sets of states are smaller. However, computing the difference with the $cBDD$ does not guarantee that the resulting BDD will be smaller. Imagine the following case: a planning task has a single fluent as goal, which means that at the layer 0 in regression we have a BDD with a single inner node. If the *mutex BDD* is used to prune unreachable states, the BDD resulting from the difference of the original BDD with the *mutex BDD* will be considerably bigger, as it will include additional information. It will represent fewer states, as the states that contain the goal fluent and violate some mutex will be effectively pruned, but it will also increase in size. This may be detrimental to the search, to the point that it may not even be able to begin if $s_\star \setminus cBDD$ is not tractably representable.

In symbolic search, the performance of the search algorithm is often heavily linked to the size of the BDDs it works with. Both memory and time (in terms of BDD manipulation) benefit from working with BDDs that succinctly represent a given boolean function. In the literature, mainly in works published by the Model Checking community, several *"don't care" minimization* algorithms have been proposed (Coudert and Madre, 1990; McMillan, 1996; Hong et al., 1997). *"Don't care" minimization* aims to succinctly represent a given function when only part of it is relevant. They receive a *function BDD* f and an additional *constraint BDD* c (also called *restrict* or *care BDD*), and aim to find a BDD g of minimum size that represents f's function in an incompletely specified way, such that $g(s) = f(s)\ \forall s \in c$.

A plausible way of exploiting *"don't care" minimization* is to assume that f corresponds to BDDs that represent sets of states and that c may be any BDD that imposes some kind of restriction over f (Edelkamp and Helmert, 2000). In our case, both the complement (that is, the negation) of the *closed list BDD* and $cBDD$ can correspond to the definition of c. Hence, using minimization algorithms instead of the conjunction may prove useful, as it may be possible to obtain smaller BDDs. The intuition behind the use of minimization algorithms is to allow representing spurious states if this means that the BDDs will be smaller. These operations are more expensive to compute than the conjunction, but if g is smaller the computation of a subsequent *image* and *pre-image* operation (which are the most expensive symbolic operations in most planning instances) may require less time.

Most BDD minimization algorithms are based on the concept of *sibling substitution*. Sibling substitution consists in replacing a node by its sibling, so that both become identical and their parent may be removed from the BDD, as illustrated in Figure 4.4. Sibling substitution requires identifying which nodes are *"don't care"* nodes: *"don't care"* nodes are all the nodes in f such that the path that leads to them evaluates to *false* in c.

The following are the minimization algorithms considered in this work:

- *constrain* (Coudert and Madre, 1990): it performs sibling-substitution recur-

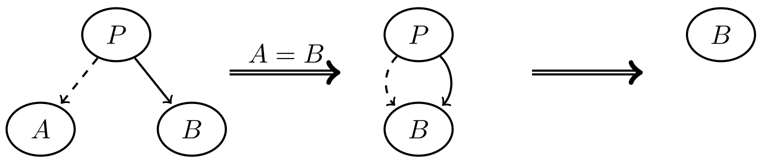


Figure 4.4: Example of sibling substitution. If node A belongs to the *don't care* set, it can be substituted by B so their parent P is removed by BDD reduction rules.

sively, replacing nodes that correspond to *don't care* values. If g is larger than f, f is returned instead. A generalization of *constrain* was also proposed under the name of *generalized-cofactor* (McMillan, 1996).

- *restrict* (Coudert and Madre, 1990): refinement of *constrain* that ensures that, whenever a node did not originally appear in f, both branches of c that stem from that node are merged. This guarantees that nodes that did not originally appear in f will not appear in g, although g may still be bigger.
- *leaf-identifying compaction* (Hong et al., 2000): it has two phases. First, it marks edges that can be redirected to a leaf node and edges whose pointed nodes are to be preserved from sibling substitution. Then, the result is obtained by redirecting edges to leaf nodes whenever possible and applying sibling substitution on non-marked edges. Thanks to the edge-marking phase, it ensures that g is smaller than f, as opposed to *restrict* and *constrain*.
- *non-polluting-and*: hybrid between regular *conjunction* and *restrict*. It performs a conjunction but, like in the *restrict* operation, whenever a node did not originally appear in f, both branches of c that stem from that node are merged.

4.4 Constrained Symbolic Abstraction Heuristics

The use of regression is not limited to backward search. For instance, Pattern Databases (PDBs) (Culberson and Schaeffer, 1998) perform regression over the goals in an abstraction of the original problem to create a lookup table that is used as a distance estimation in the original problem. PDBs in explicit-search that make use of mutexes are known as Constrained PDBs (Haslum et al., 2007). Constrained PDBs prune transitions that go through abstracted states that violate the constraints. Thus, they may potentially prune spurious paths (solution plans in the abstract state space that do not have a corresponding plan in the original problem) and strengthen the derived heuristic. A symbolic version of PDBs for their use in symbolic search has also been proposed (Edelkamp, 2002), so in this work we propose a constrained version of symbolic PDBs.

Symbolic Constrained PDBs exploit constraints in regression either by encoding them in a $cBDD$ or by using constrained TRs, as we studied in the previous subsections. However, there are some subtleties related to the usage of constraints in an abstract search space that make it different from backward search in the original problem. PDBs ignore some variables of the problem. Hence, constraints that refer to variables not in the pattern are never violated and cannot be taken into account to prune abstract states. Haslum overcame this problem by individually checking abstracted variables in the preconditions of the operators, so there would not exist transitions corresponding to operators that could lead to spurious states in regression. However, operators in symbolic search may be merged (Torralba et al., 2013a), which makes individually checking them not possible.

Nevertheless, there are alternatives to Haslum's method. In particular, the use of *e-deletion* to modify the applicability of operators in regression, described in Section 3.4.2, allows avoiding the generation of spurious states even when a subset of the variables has been abstracted away. In Section 4.2.3 we proposed encoding constraints in the preconditions of the TRs to avoid the generation of spurious states in regression. For this, we assumed that the parent set of states did not contain spurious states. In abstractions this cannot be ensured anymore, so we must also encode the *e-deleted* fluents in the effects. These *e-deleted* fluents act like negative preconditions in regression, so operators that would generate spurious states become non applicable. This way, encoding constraints in the preconditions of the TRs and *e-deleted* fluents in the effects allows obtaining cPDBs more informed than the ones obtained from TRs with constraints in the preconditions only and from the difference with $cBDD$.

4.5 Experimentation

In this section we analyze the impact of using constraints and BDD minimization in different settings, ranging from symbolic blind search to the generation of symbolic abstraction heuristics. Our motivation is to check whether h^2 mutexes and invariant groups improve over the constraints inherent to the SAS$^+$ formulation of the problem and to see if more complex BDD manipulation pays off.

h^2 was implemented on top of Fast Downward. For symbolic blind search and symbolic A* (BDD-A*) with PDBs we use the Gamer planner (Kissmann and Edelkamp, 2011). As each planner uses its own SAS$^+$ variables, mutex fluents are extrapolated to Gamer's SAS$^+$ encoding. All the experiments with Gamer use the same variable ordering, which is optimized prior to the search.

Results with different methods to prune spurious states are reported, sharing the same nomenclature in all the subsections. $cBDD$ contains constraints from both h^2 mutexes and invariant groups. $\mathcal{M}_\emptyset$ is the baseline version which does not use $cBDD$. $\mathcal{M}^\&$ computes the conjunction with the negation of $cBDD$. The four aforementioned BDD-minimization algorithms are used: *don't care minimization*($\mathcal{M}^{dcm}$), *restrict*($\mathcal{M}^{res}$), *constrain*($\mathcal{M}^{con}$) and *non-polluting-and*($\mathcal{M}^{np\&}$). Finally *e-del* is the version in which the TRs take into account mutex and invariant group constraints instead of using $cBDD$. Results are reported with the regular set of operators ($\mathcal{O}$) and with the set of operators after pruning spurious operators found by disambiguating their preconditions ($\mathcal{O}^-$).

We run experiments on the benchmarks of the optimal track of the International Planning Competition 2011[1]. All our experiments were run and validated with the IPC-2011 software on a single core of an Intel(R) Xeon(R) X3470 processor at 2.93GHz. The experimental setting is the same as in IPC-2011: 1800 seconds per problem and 6GB of available memory. Time score and coverage follow the same rules as in IPC-2011 too. BDD operations are implemented using Fabio Somenzi's CUDD[2] 2.5.0 library. For the *image* and *pre-image* computation we used a disjunctive partition of the TRs, merging the TR of each operator up to a maximum size of 100000 nodes. This method is simple and has proved to be more efficient than other approaches (Torralba et al., 2013a). Similarly, if a single $cBDD$ surpasses 100000 nodes, it is also encoded as several smaller ones.

In *Elevators* and *Transport* neither additional h^2 mutexes nor constraints from invariant groups were found. In *Floortile*, *Nomystery*, *Openstacks*, *Parcprinter*, *Pegsol* and *Woodworking* $cBDD$ had fewer than 10000 nodes in all the problems. More than one individual $cBDD$ were needed in some instances of the following domains (number of BDDs in the worst case between parentheses): *Barman*(2), *Parking*(25), *Scanalyzer*(10), *Sokoban*(13), *Tidybot*(8) and *Visitall*(6). Overall neither the size of $cBDD$ nor the time spent pruning unreachable states was significant. Note that the size of $cBDD$ also depends on the order of the variables: an order more suitable to their representation might reduce their size.

When encoding the constraints in the TRs (*e-del* configuration), the sizes of the TRs vary. In *Barman*, *Tidybot* and *Visitall* the TRs actually become smaller in most instances. In *Openstacks*, *Pegsol* and *Woodworking* the size is roughly the same ($\pm 10\%$). In *Floortile* the TRs grow by more than 50%, in *Nomystery* and *Parcprinter* they become several times bigger and in *Sokoban* and specially in *Scanalyzer* the number of individual TRs needed to encode the operators grows by a significant amount. The worst case is *Parking*, in which the size of the TRs of the individual operators blows up when the constraints are added and exceed the available memory even before beginning the search. This is due to the high number of mutexes that are found in this domain, as all variables in *Parking* interact heavily with each other.

4.5.1 Symbolic Unidirectional Blind Search

First we start with the simplest case, backward blind search. We compare the performance of symbolic backward search with and without constraints against symbolic forward search. Table 4.1 shows the time score comparison of different configurations of forward and backward blind search.

The impact of pruning spurious operators ($\mathcal{O}^-$) is small except in *Tidybot* and sometimes does not compensate the time spent computing h^2 if this is only done to

[1]http://www.plg.inf.uc3m.es/ipc2011-deterministic
[2]http://vlsi.colorado.edu/ fabio/CUDD

	BACKWARD									FORWARD	
	$\mathcal{M}_\emptyset$		$\mathcal{M}^{\&}$		$\mathcal{M}^{dcm}$	$\mathcal{M}^{res}$	$\mathcal{M}^{con}$	$\mathcal{M}^{np\&}$	*e-del*	$\mathcal{M}_\emptyset$	
	$\mathcal{O}$	$\mathcal{O}^-$	$\mathcal{O}$	$\mathcal{O}^-$	$\mathcal{O}^-$	$\mathcal{O}^-$	$\mathcal{O}^-$	$\mathcal{O}^-$	$\mathcal{O}^-$	$\mathcal{O}$	$\mathcal{O}^-$
Barman	0.00	0.00	5.69	5.98	1.13	1.58	1.64	5.91	**9.29**	7.35	7.47
Elevators	2.48	2.49	2.48	2.48	2.48	2.49	2.49	2.49	2.49	**15.88**	15.82
Floortile	5.06	5.18	13.25	**13.60**	10.02	11.89	12.60	13.41	13.39	0.71	0.73
Nomystery	10.46	10.96	10.21	10.95	10.48	10.61	10.41	10.94	**11.04**	9.63	9.33
Openstacks	15.30	14.68	16.19	15.97	7.59	10.75	11.47	15.85	15.95	**19.77**	18.22
Parcprinter	4.23	4.18	15.14	15.27	12.04	13.54	13.60	14.69	**15.48**	5.31	5.55
Parking	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
Pegsol	0.00	0.00	2.07	1.92	0.85	0.85	0.85	1.92	2.20	**16.74**	16.38
Scanalyzer	8.72	8.48	8.38	8.43	3.08	3.65	3.62	8.45	**9.00**	8.49	8.20
Sokoban	0.27	0.27	16.48	**16.68**	11.15	12.35	14.50	16.12	16.64	15.87	14.96
Tidybot	0.00	1.00	2.71	6.65	0.87	0.93	0.87	4.74	6.80	10.05	**13.92**
Transport	1.79	1.79	1.79	1.79	1.79	1.79	1.79	1.79	1.79	**6.00**	5.85
Visitall	8.51	8.51	8.52	8.52	8.43	8.52	**8.77**	8.52	8.57	7.62	7.60
Woodworking	10.76	9.83	17.43	17.69	8.47	8.95	9.03	17.13	**18.68**	5.31	4.97
Total	67.58	67.37	120.34	125.92	78.40	87.89	91.64	121.96	**131.32**	128.73	128.99
COVERAGE	92	94	144	148	113	121	122	145	**150**	149	149

Table 4.1: Time score and total coverage of unidirectional blind search.

prune such operators (forward and backward $\mathcal{M}_\emptyset$). As expected, it benefits backward search more than forward search, as spurious operators may be applicable in regression. Additionally, in versions that already compute h^2 for other purposes, the score with $\mathcal{O}^-$ stays roughly the same or improves, so operator pruning is always recommended in these cases.

Using constraints to prune spurious states improves the results by a very significant margin, almost doubling the time score and solving 50% more problems overall. In *Barman*, *Sokoban*, *Tidybot* and *Pegsol* almost no search was accomplished by backward search without mutexes, while pruning spurious states allows backward search to solve some problems in those domains. For example, the maximum g-layer expanded backwards without mutexes in the first problem of *Barman* is five, whereas with mutexes the optimal solution, whose cost is 90, is found. Also, although computing $cBDD$ requires some time and memory, it does not harm in any domain.

The reported results include the use of invariant group constraints in all the configurations that use constraints. When disabling the use of these constraints the same coverage is obtained, although the time score worsens perceptibly in *Sokoban* and slightly in *Woodworking* and *Openstacks*, losing six points overall with the $\mathcal{M}^{\&}$ configuration.

BDD minimization is not useful in this setting. This is because reducing the BDD that represents the set of predecessor states by including some spurious states often means that the set of successor states is bigger. This has no impact in terms of memory, as the set of successor states can be minimized afterwards in the same way, but it affects negatively the time required to compute the *pre-image* (apart from requiring more time than a regular conjunction with the $cBDD$). Overall and although in some cases a memory reduction of up to a third for some BDDs is obtained, the extra time does not pay off. Thus, $\mathcal{M}^{\&}$ dominates all the BDD minimization configurations in all domains.

Finally, encoding mutexes and invariant group constraints in the TRs instead of using *cBDD* is the most efficient version. This is because *e-del* directly generates BDDs that do not contain spurious states, instead of generating a potentially much bigger BDD with spurious states and intersecting it with *cBDD* afterwards. Surprisingly, the size of the TRs does not affect the performance, which means that the performance of *pre-image* seems to depend on the size of the resulting BDD rather than on the size of the predecessor BDD and the TRs.

On a per domain comparison we can observe that the directionality of the domains has a huge impact on the performance of the planner (Massey, 1999). This hints that a bidirectional approach is probably more efficient than unidirectional search, which is explored in the following subsection.

4.5.2 Symbolic Bidirectional Blind Search

After assessing the viability of constraints in isolation, we now use them in a state-of-the-art symbolic bidirectional blind version of Gamer. Spurious operator pruning is enabled in all the configurations. The only other modification with respect to the version of Gamer used in (Torralba et al., 2013a) is that we dynamically check the time and memory consumed per step. During a step, if it uses more than twice the memory or time than the last step in the opposite direction, we interrupt it and switch the direction of the search. This makes $\mathcal{M}_{\emptyset}$ solve four fewer problems in *Tidybot* (7 instead of 11), three of which are recovered thanks to spurious operator pruning. We exclude BDD-minimization methods because, as shown in the unidirectional case, they are not useful in the symbolic blind search setting. Thus, we only compare the baseline with the version that uses *cBDD* ($\mathcal{M}^{\&}$) and the one that uses TRs with constraints (*e-del*), both of them also with invariant group constraints.

Table 4.2 shows that symbolic bidirectional blind search is able to improve over both forward and backward search. As in the unidirectional version, a consistent improvement is obtained when using constraints, increasing the coverage from 172 problems to 199 with *e-del*. *Nomystery* is the only domain where applying mutexes actually makes the search slightly slower, although this does not affect coverage. $\mathcal{M}^{\&}$ and *e-del* yield similar results in most domains (a difference in score smaller than 0.5 points), although an increase in performance is obtained by *e-del* in *Barman*, *Parcprinter*, *Scanalyzer* and *Woodworking*.

Overall the results of the bidirectional blind search version of Gamer with constrained TRs are remarkable, solving 14 more problems than the winner of IPC 2011, Fast Downward Stone Soup (Helmert et al., 2011).

	$\mathcal{M}_\emptyset$		$\mathcal{M}^{\&}$		*e-del*	
Barman	5.46	8	9.78	11	**12.00**	**12**
Elevators	18.13	**19**	**18.17**	**19**	17.92	**19**
Floortile	5.89	10	**13.56**	**14**	13.46	**14**
Nomystery	**16.00**	**16**	15.06	**16**	15.11	**16**
Openstacks	17.68	**20**	18.93	**20**	**19.16**	**20**
Parcprinter	5.79	8	14.09	15	**15.37**	**16**
Parking	**0.00**	**0**	**0.00**	**0**	**0.00**	**0**
Pegsol	14.65	17	**18.42**	**19**	18.38	**19**
Scanalyzer	8.52	**9**	8.28	**9**	**9.00**	**9**
Sokoban	13.63	**19**	**18.49**	**19**	18.21	**19**
Tidybot	8.98	10	**15.53**	**16**	15.33	**16**
Transport	8.67	**9**	**8.83**	**9**	8.67	**9**
Visitall	**10.98**	**11**	10.95	**11**	10.84	**11**
Woodworking	10.81	16	17.74	**19**	**18.47**	**19**
Total	145.17	172	187.83	197	**191.94**	**199**

Table 4.2: Time score and coverage of bidirectional blind search.

4.5.3 BDD-A* with PDBs

After observing the positive impact of constraints on symbolic blind search, we analyze the case of symbolic abstraction heuristics, in which symbolic backward search is performed in an abstracted state space. Our goal is testing whether the constrained versions of symbolic abstraction heuristics return higher estimates than the unconstrained versions. We performed experiments with BDD-A* guided by symbolic PDBs. The choice of variables for the patterns is the same as the one proposed by Kissmann and Edelkamp (2011). Table 4.3 shows the results. First, pruning spurious operators allows solving four additional problems, one in *Barman* and three in *Tidybot*. When using mutexes, the coverage goes up to 184 problems with both $\mathcal{M}^{\&}$ and *e-del*. The use of BDD-minimization operators performed slightly worse than $\mathcal{M}^{\&}$ and *e-del*, so we did not include them for conciseness. The time score was also left out because BDD-A* spends half of the available time computing the PDBs, which skews the time score and makes it not as representative.

Comparing the results of Tables 4.1 and 4.3 we can see that constraints have a smaller impact when using abstractions. PDB heuristics are only useful in *Parking*, *Sokoban* and *Tidybot*, where an additional problem per domain is solved. Constraints helped more in cases in which backward blind search was not feasible at all, as the number of problems in which a single step backward is not possible is reduced when using them. Overall, bidirectional blind search performs almost as good or better than BDD-A* with symbolic PDBs, as it can exploit the directionality of the planning instances better.

We can conclude that the pruning power of constraints is reduced considerably in abstracted spaces: the constraints in which abstracted variables appear are of no use and the abstracted space is less constrained than the original one, so there is less margin for improvement. Nevertheless the use of constraints never hurts and is in all cases equal or better than the configuration with no mutexes, so there is no

	$\mathcal{M}_\emptyset$ ($\mathcal{O}$)	$\mathcal{M}_\emptyset$ ($\mathcal{O}^-$)	$\mathcal{M}^{\&}$	*e-del*
Barman	6	7	**8**	**8**
Elevators	**19**	**19**	**19**	**19**
Floortile	12	12	**14**	**14**
Nomystery	**14**	**14**	**14**	**14**
Openstacks	**20**	**20**	**20**	**20**
Parcprinter	**9**	**9**	**9**	**9**
Parking	**1**	**1**	**1**	**1**
Pegsol	**17**	**17**	**17**	**17**
Scanalyzer	**9**	**9**	**9**	**9**
Sokoban	**20**	**20**	**20**	**20**
Tidybot	14	**17**	**17**	**17**
Transport	**6**	**6**	**6**	**6**
Visitall	**11**	**11**	**11**	**11**
Woodworking	**19**	**19**	**19**	**19**
Total	177	181	**184**	**184**

Table 4.3: Coverage of Symbolic PDBs + BDD-A*.

reason why constraints should not be used.

4.6 Discussion

In this chapter we showed the relative pruning power of h^2 mutexes in symbolic search, proving that the constraints encoded in the SAS^+ formulation do not suffice to detect a significant amount of spurious states in many domains. Additional constraining techniques were successfully employed, and the impact of BDD-minimization operations that work with constraining BDDs was tested. Note that h^2 mutexes were implicitly used before (Jensen et al., 2006), although the reported results did not show a significant increase in performance.

The results seem to contradict the assumption in planning that progression is more robust than regression, at least in optimal symbolic search. Previous results on both symbolic and explicit-state search suggested that forward search outperformed backward search in the IPC benchmarks. For example, in (Torralba et al., 2013a) it is reported that the percentage of forward search performed by Gamer with bidirectional blind search is greater than 90% for 6 out of 14 domains and 75% overall. Only in *Floortile* and *Woodworking* backward search was superior to forward search. Another example is Patrik Haslum's forward version of HSPr in the IPC 2008[3], which was as good or better than the original backward version in all the domains except *Scanalyzer*. However, the results shown in Table 4.1 change this picture: when using h^2 mutexes, the results of backward and forward symbolic search are close, with a relatively high degree of variability between different domains.

An important conclusion to be drawn is that symbolic search seems to work

[3]http://ipc.informatik.uni-freiburg.de/Results

better in regression than explicit-state search. The underlying reason is subsumption of states in regression. This occurs when the set of fluents that compose a partial state is a subset of the set of fluents of a newly generated one. In this case the latter should be reported as a duplicate of the former, which is seamlessly detected when using a symbolic closed list but which is not trivially detected when using a closed list in Disjunctive Normal Form, as it happens in explicit-state search. Similarly, the detection of the collision of frontiers in bidirectional search is trivial if the backward search is symbolic (the forward search can be either symbolic or explicit-state), which tips the scales further in favor of symbolic search in regression.

Regarding the good performance of symbolic bidirectional blind search, the impact of the directionality of the domains (Massey, 1999) explains why this configuration fares so well, as a simple alternating strategy allows choosing the direction in which the problem may be more easily solved. We leave the implementation of a bidirectional version of BDD-A* that uses constraints as future work. We also plan on investigating whether variable orders derived from the constraints may be more useful than the range of orderings tested in the literature (Kissmann and Hoffmann, 2013). Such orderings may provide a more concise representation of the set of spurious states, which could overcome the drawbacks of using constraints in domains like *Parking*, in which $cBDD$ becomes too large to be manageable.

Chapter 5

Backwards Generated Goals

Heuristic search with reachability heuristics is arguably the most successful approach in Automated Planning up to date. In addition to an estimation of the distance to the goal, the relaxed plans obtained with such heuristics provide the search with useful information, like preferred operators and look-ahead states. However, this information is extracted only from the beginning of the relaxed plan. In this chapter, we propose using information extracted from the last actions in the relaxed plan to generate intermediate goals backwards. This allows us to use information from previous computations of the heuristic and reduce the depth of the search tree, both in the original space and in the computation of the heuristic.

5.1 Introduction

The use of reachability heuristics along with a forward search algorithm has proved to be one of the most effective approaches in Automated Planning, as seen in the last International Planning Competitions. Most heuristics of this kind are based on the relaxation of the delete effects of the actions. In particular, the FF heuristic (also known as the *relaxed plan heuristic*) first used in Fast Forward (Hoffmann and Nebel, 2001) still remains a very effective way of guiding a forward search in the state space.

Apart from the heuristic numeric value obtained, the actions that compose the relaxed plan offer additional information that can be exploited in the search process. Helpful actions (Hoffmann and Nebel, 2001) can be used to prune unpromising successors or to discriminate between successors by assigning them to different priority queues (Röger and Helmert, 2010); look-ahead states (Vidal, 2004b) are an attempt to advance several steps at once in the search state by combining and executing applicable actions of the relaxed plan; and macro-actions (Botea et al., 2005) may be inferred online from the relaxed plan to redefine the domain and reduce its depth.

A common aspect of these techniques is that they all take into account only the first actions of the relaxed plan. This is reasonable, given that these techniques require legal sequences of actions in the evaluated state, and relaxed plans have a higher probability of including illegal or non-applicable actions the farther the actions are from the evaluated state. However, this ignores potential additional information the relaxed plan may contain.

In this work we propose a novel method to take advantage of information from the last actions of the relaxed plan. The motivation behind this work is that the last actions of the relaxed plan are often similar across different calls to the heuristic function. This fact allows the planner to improve the search by: reusing information from previous computations of the heuristic; or getting more accurate heuristic estimations. The way of exploiting this observation is by generating intermediate goals backwards by applying in regression actions from the last part of the relaxed plan. This way, in subsequent computations of the heuristic the closest intermediate goal to the current state can be detected and the estimation of the distance can be computed to the detected intermediate goal and not to the original goal. We have called this approach the Backwards Generated Goals (BGG) heuristic.

BGGs provide additional information that can be used to reduce the number of steps taken in the reachability heuristic. Also, this information can be used to derive more accurate heuristic estimations for two reasons: first, BGGs will usually be closer to the evaluated state than the original goal, and reachability heuristics tend to be more accurate as the search gets closer to the goal because, if the relaxed exploration requires fewer steps, there is less margin for error to accumulate. Also, there exist the possibility of adding the estimation of the heuristic to the distance of the detected BGG to the original goal if a more accurate estimation *to the original goal* is required. Furthermore, the use of BGGs allows stopping the search earlier when satisfying an intermediate goal, as the path to the original goal can be built by tracing back the generation of the intermediate goal. This relies on a careful generation of extra information together with the intermediate goal.

To ensure the validity of the backwards generated goals, actions must legally support the reached goals. This is done by using concepts from regression like *e-deletion*: considering delete effects when supporting goals and taking into account static mutexes between the preconditions of the actions and the goal propositions not satisfied by the supporting actions. Experimental evaluation of the techniques presented here shows improvements in coverage and number of expanded nodes over the regular FF heuristic. In particular, the approach seems to improve performance substantially in traditionally hard domains to common reachability heuristics.

5.2 Reachability Heuristics and the FF Heuristic

When using a heuristic search algorithm, the estimation of the distance to a goal state is often obtained by solving the same problem, but after introducing some kind of relaxation. The relaxed plan heuristic originally made use of the planning graph implemented by Graphplan (Blum and Furst, 1997). It builds a layered directed graph alternating facts and actions until all goals appear at some level, not taking into account mutexes between both actions and propositions and ignoring delete effects. Figure 5.1 shows a relaxed planning graph in which circles are propositions of S and squares are actions of A. Propositions and actions are organized in vertical layers. Edges that connect an action to propositions of the previous level mean that those propositions are preconditions of the action. Edges that connect an action to propositions of the next level mean that those propositions are *adds* of the action. Edges that connect propositions between proposition layers are *no-op* actions, used to propagate already achieved propositions forward. The propositions highlighted in pink are the goals of G.

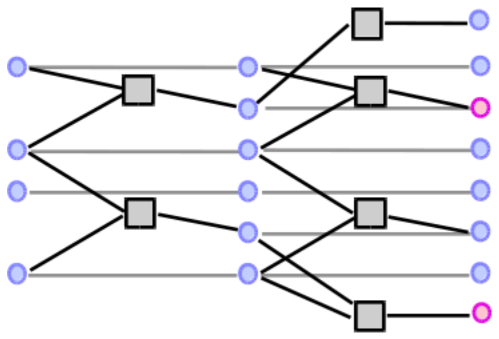


Figure 5.1: Illustration of a relaxed planning graph.

Once the graph has been generated and all the goals have been achieved at some level, a plan is extracted by a backtrack-free search algorithm in which actions that make the goals true are greedily selected. The supported goals are removed and the preconditions of the actions are added to the goal set, until all the goals belong to the initial state. The actions are chosen taking into account the level they first appeared, choosing those with lower levels. The heuristic value returned is the number of actions in the relaxed plan. As presented in Section 3.5, in most modern planners the reachability analysis algorithm is implemented as uniform-cost search rather than as a relaxed planning graph because it is more efficient and it allows a more formal characterization of the algorithm and the heuristics. However, in this chapter we will use the traditional relaxed planning graph nomenclature, as it will help understanding how BGGs work.

While planning as heuristic search is the paradigm used by most successful state-of-the-art planners, it has some disadvantages. Even though reachability heuristics can be computed in polynomial time, they still require heavy computation. Besides, due to the complex interactions that often appear in planning problems, heuristics must be recomputed for every state from scratch. While this has been addressed by earlier works (Refanidis and Vlahavas, 2004; Liu et al., 2002), reducing the impact of this heavy computation required for heuristic evaluation still remains an open question.

One way to alleviate this is to extract additional information from the heuristic computation. Reachability heuristics solve a relaxed version of the problem, and thus yield a relaxed plan as a solution. The relaxed plan may contain actions in common with a possible solution, which can favorably bias the search. Preferred operators and look-ahead states are the most prominent examples of this kind of techniques. In particular, helpful actions are sometimes more relevant than the accuracy of the heuristic used (Richter and Helmert, 2009). Since these actions must have the possibility of belonging to a real solution, they must be legal actions even in the relaxed case. This is ensured by checking that the action is applicable in the evaluated state in the case of helpful actions, and that none of the preconditions of the actions that lead to the look-ahead state is deleted by another action. Therefore, this limits the actions to those close to the evaluated state.

However, as shown in Section 3.2, it is possible to ensure the consistency of the actions with the set of goals and thus guarantee their applicability in regression by using *e-deletion*. This gives the opportunity to extract similar information from actions far from the beginning state. In forward state search, goals are static. Because of this, in many cases the last actions that lead to the goal in the relaxed plan will be the same for different evaluated states - in fact, they may even belong to a solution plan -. Hence, generating sets of intermediate goals as done in backward search using the information derived from the relaxed plan may allow reusing information from relaxed plans obtained in the evaluation of previously evaluated nodes. This can benefit the search in essentially in two ways: first, it may decrease the depth of the search in both the heuristic computation and the forward state search; second, it may capture additional information from constraints that may appear in areas of the search space around the goal.

5.3 Intermediate Goals and Reachability Heuristics

A reachability analysis is actually a uniform-cost search in a delete relaxation of P. There is an important observation about this: uniform-cost search exhibits a wave-like behavior. Therefore, it not only gives an estimation of the distance to a goal, but also estimates that the first reached goal is the closest to the state in case multiple goal states are present. To better illustrate this, let us consider the example shown in Figure 5.2. There is a grid in which some cells are labeled as goal cells, and a key is needed to open a chest that contains the gold. There is only one key, held by the agent in the initial state. An agent can only move to adjacent (non-diagonal) cells, and there are dark cells. The key is lost when going through dark cells. In this case, in the heuristic computation the first level of the relaxed planning graph will check whether any of the cells at distance one is a goal cell; at the second level, all the cells at distance 2 will be checked, and so on. The heuristic computation will find the closest goal cell in the relaxed problem, independently of how many goal cells exist in the grid, and will stop. The relaxed plan may be illegal (if for example there is a dark cell in the way to a goal cell, since the key would be lost), but it still gives an estimation to the closest goal.

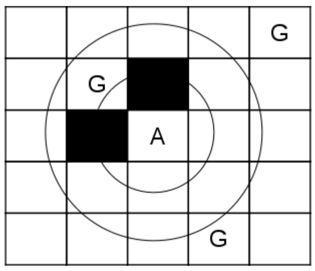


Figure 5.2: The agent is located in A and cells marked with G are goal cells. In the heuristic computation, only two levels are expanded estimating that the goal cell to the left and above the agent is the closest one.

In most domains the set of goal propositions does not describe a complete state - an exception being the *N-Puzzle* domain, for instance - so the reachability analysis already deals with multiple goal states implicitly. The set of goal propositions G does not describe a complete state when the value of some proposition in the instantiated problem is not defined. For example, if S=$\{a,b\}$ and G=$\{a\}$, both s_1=$\{a,b\}$ and $s_2 = \{a, \neg b\}$ are goal states. However, this fact does not change the behavior of the heuristic, meaning that regular reachability heuristics effectively take into account multiple potential goals. This interesting property can be extended to the case in which there are different sets of goal propositions as well.

Taking advantage of this fact, the main contribution of this work is to generate multiple intermediate goals that lead to the original goal. Actions known to lead to the problem goals are used to generate the intermediate goals. To formally represent the existence of multiple set of goals a change on the definition of the problem is needed: the goal is not G anymore but rather a set of sets of propositions $\mathcal{G}$ such that initially $\mathcal{G} = \{G\}$. At each call to the heuristic function, a new intermediate goal set G' is obtained so it can be added to the set of goals $\mathcal{G}$. G' is generated by choosing an action $a \in A$ from the relaxed plan that supports one of the goal propositions $g \in G$ such that $G \in \mathcal{G}$. G' is generated by applying a in G in regression as described in Section 3.2. This means that we remove the propositions in G supported by a and add pre(a) to the aforementioned set of propositions. This new set G' is then added to $\mathcal{G}$, a list of goals relevant to subsequent computations of the heuristic.

Every BGG $\in \mathcal{G}$ stores the chosen action $a \in A$ and the distance to the original goal G for two purposes: allowing reconstructing a path by tracing back the chosen actions; and obtaining the distance from the intermediate goal to the original goal, that is, the cost of the intermediate goal. Optimality on this distance cannot be ensured, as there may be shorter paths from the intermediate goal to the original goal G. Having several sets of goal propositions also changes the stopping criterion of the reachability analysis: instead of stopping the expansion of the relaxed planning graph when all goal propositions $g \in G$ appear in a given layer $P_i \subseteq S$, the expansion stops at a propositional layer P_i when $\exists G_j \in \mathcal{G}$ such that $G_j \subseteq P_i$. Figure 5.3 shows how the aforementioned wave-like behavior of reachability heuristics allows stopping the reachability analysis earlier if a BGG is satisfied. In this case the circles around the evaluated state S_i represent the layers of the relaxed planning graph. As we can see, once the reachability analysis satisfies BGG_j it stops even if it has not reached G. The relaxed plan is then extracted by tracing back actions from BGG_j.

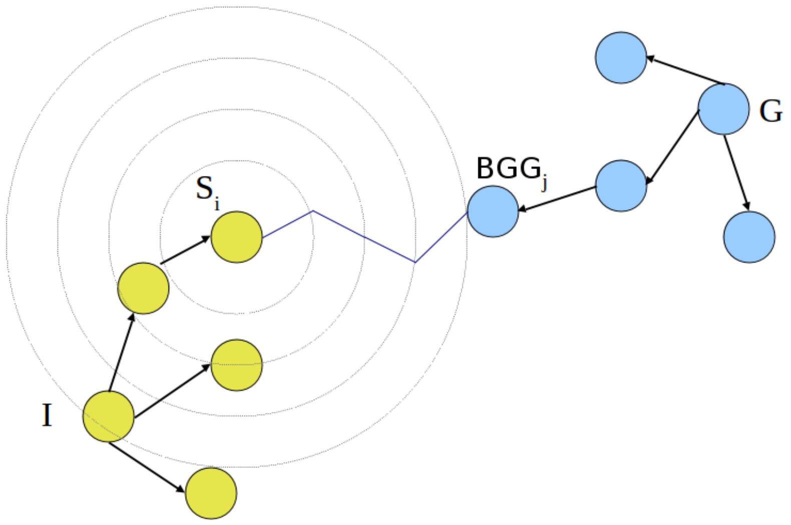


Figure 5.3: Computation and extraction of a relaxed plan from the evaluated state S_i to BGG_j.

The implementation of this technique is closely related to how actions are known to be applicable at a given level in the implementation of heuristic search planners like FF and Fast Downward. The relationship is clear, as the preconditions of any action $a \in A$ form subgoals themselves. Each proposition $p \in S$ maintains a list of indexes of the sets of goals $G_i \in \mathcal{G}$ they appear in, that is, p points to G_i if $p \in G_i$. Also, we define a counter that stores the number of unsatisfied propositions of each intermediate goal set G_i. Whenever a goal proposition p is first achieved by an action in the relaxed planning graph, the counter of the goal sets $G_i \in \mathcal{G} \mid p \in G_i$ is decreased by one. When the counter of any G_i reaches zero, it is guaranteed that there is a relaxed plan that can reach G_i and so the reachability analysis can stop.

A key difference with respect to the standard computation of the FF heuristic however is that we require at least one action that reached the intermediate goal set in the last action layer of the relaxed planning graph to be a *legal support*. The concept of legal support is based on the application of the action in the search space: it must be able to appear as the last action in a valid solution plan. This is the same case presented in Section 3.5.1, in which in order to ensure that at least there will be one applicable preferred operator in regression one of the supporters of the set of goal propositions must be *consistent*. Therefore, the conditions that must be fulfilled to ensure that a supporting action $a \in A$ of some $G_i \in \mathcal{G}$ is a legal support of G_i and is able to generate a new BGG G' are:

- a is consistent with G_i following Definition 12; and
- the new set of goal propositions G' must not have been previously generated ($G' \notin \mathcal{G}$)

The first constraint is straightforward: an action a that is not *consistent* with a set of goal propositions G_i can never be the last action of a plan, as it is not applicable in regression in G_i. In fact, this constraint usually ensures that there is a legal path from a given intermediate goal to the original goal, as it avoids the generation of spurious sets of goal propositions detectable by the use of mutexes.[1] It is also interesting to note that enforcing the consistency of at least one goal supporter could also be done for the original FF heuristic. In this case at least one legal supporter for a goal proposition $g \in G$ would be required, which would turn the FF heuristic into a semi-relaxed one using a quite plausible criterion.

The basic idea of the second constraint is that, whenever a duplicated goal is generated, that goal was necessarily already supported in the relaxed graphplan at an earlier lever. Then, if the heuristic computation did not stop at that earlier level when supporting that already generated goal, it must be an unreachable goal at that level as no legal action was found for it so far. This can be explained going back to the leveled approach of the heuristic computation. Intermediate goals are generated by taking away the supported propositions and adding the actions preconditions. Preconditions must be satisfied at earlier levels than the propositions the action supports. If the new set of propositions that includes the preconditions of the action was already an intermediate goal, this means that this goal had already been satisfied at earlier levels of the heuristic computation. This is so because the already existing intermediate goal only differs from the reached goal in the preconditions and effects of the action, and these preconditions belong to earlier levels.

5.3.1 Implementation Details

It is possible for a given set of goals $G_i \in \mathcal{G}$ to be legally supported in several ways, which opens the possibility of generating several BGGs. However, in our implementation we have chosen to generate a single BGG per evaluated state to avoid having having an excessive number of BGGs. Hence, one action must be selected. First, a goal proposition among those legally supported must be chosen.

[1]Note that it is in fact possible to generate spurious BGGs, just like any other set of propositions generated in regression. Using disambiguation may help reduce the number of spurious BGGs in some domains, although spurious BGGs undetectable by mutexes are not frequent and even if they are spurious they may still help during the computation of the heuristic - they will never be satisfied in the original search space.

In the current implementation, propositions are heuristically chosen by the level at which they first appear, preferring propositions farther from the initial state. The intuition behind this decision is that these propositions seem more difficult to satisfy and thus they are probably achieved in later stages of the solution plan. This concept is loosely related to the way actions are heuristically chosen when extracting the relaxed plan, although in a reversed way. Second, if the proposition has several supporting actions, those appearing closer to the initial state are chosen first, as their preconditions are deemed easier to achieve. The reachability analysis is described in Algorithm 1, while Algorithm 2 describes the way a legal action is sought and chosen to generate a new intermediate goal.

Algorithm 1: Heuristic Computation with Intermediate Goals.

```
Data: Current State $s \subseteq S, \mathcal{G}$
Result: Intermediate Goal igoal
begin
    SatisfiedPropositions $\longleftarrow s$
    while not leveledOut($\mathcal{G}$) do
        NewlySatisfied $\longleftarrow \emptyset$
        foreach $p \in$ SatisfiedPropositions do
            foreach GoalReference $\in p$ do
                Unsatisfied $\longleftarrow Unsatisfied - 1$
                if Unsatisfied = 0 then
                    igoal $\longleftarrow$ LegalGoal(GoalReference)
                    if igoal $\neq$ NULL then
                        return igoal
            foreach ActionReference $\in p$ do
                Preconds $\longleftarrow$ Preconds-1
                if Preconds = 0 then
                    add Effects to NewlySatisfied
            SatisfiedPropositions $\longleftarrow$ NewlySatisfied
    return NULL
end
```

As depicted in Algorithm 1, it is possible for the relaxed plan graph to level out before a supporting action is found. This is due to the constraints on the legality of the supporters. In this case, the goals are unreachable and the state is then successfully detected as a dead end.

Intermediate goals also affect the forward search stopping criterion. Because of the first constraint on the legality of the supporters, a path can be built from any intermediate goal to G by tracing back the actions used in the intermediate goal generation. Consequently, once the forward search satisfies a goal, be it an intermediate one or G, the search stops and returns a valid plan. Stemming from this fact, two different heuristic values can be extracted. Since intermediate goals

Algorithm 2: LegalGoal Function.

```
Data: Intermediate Goal $igoal$
Result: New Intermediate Igoal $igoal$
begin
    /* Propositions are sorted by the level they were satisfied in,
       those at later levels first                                  */
    foreach Proposition $\in$ igoal do
        foreach SupportingAction $\in$ Proposition do
            newIgoal $\longleftarrow$ igoal
            take out AddEffects from newIgoal
            if SupportingAction deletes newIgoal then
                continue
            if IsMutex(Preconds, newIgoal) then
                continue
            add Preconds to newIgoal
            if newIgoal already exists then
                continue
            return newIgoal;
    return NULL;
end
```

store the distance to G g_{bgg}, the value extracted in the heuristic computation may be either the distance to the intermediate goal h_{bgg} or $h_{bgg} + g_{bgg}$, in which h_{bgg} is the length of the relaxed plan extracted from that intermediate goal. If the main objective is finding a solution regardless of its quality, h_{bgg} is good enough, as the forward state search only has to reach an intermediate goal to build a valid plan. If plan quality is important, adding both values may give a more accurate estimation than the regular FF heuristic thanks to delete effects being taken into account to some degree along the path from the reached BGG to G. Just like the FF heuristic, both cases can be extended to a metric different from plan length too, being cost the most common one.

5.4 An Example of Backwards Goal Generation

To show how the process of backwards generating goals works, we will include an example from a challenging domain. The domain is the *Gold-Miner* from the learning track of the International Planning Competition held in 2008[2]. This is a domain similar to the aforementioned grid domain. The goal in this case is picking up the gold, which appears at only one cell. On the grid there are rocks that block the way, that can be either hard or soft. There is a laser and unlimited bombs which can be used to clear the rocks. Soft rocks can be destroyed with either one, but hard rocks require the laser. Besides, when using a bomb it is lost and another one

[2]IPC 2008 website: http://ipc.informatik.uni-freiburg.de/

must be picked up from the pile of bombs if needed. The drawback of the laser is that it deletes the gold proposition if used to clear up the way in the cell where the gold is at, leading to a dead end. Figure 5.4 shows what a problem in *Gold-Miner* looks like.

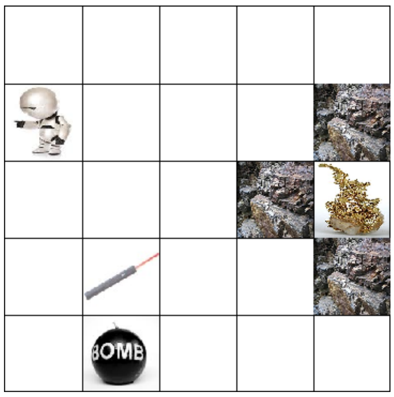


Figure 5.4: Example of a problem in *Gold-Miner*.

The difficulty of this domain lies in the fact that, since delete effects are ignored, using the laser has no drawbacks when solving the relaxed problem. Therefore, once the laser is picked up (which is often mandatory to clear the way through a hard rock) picking up a bomb to destroy the last rock is almost never considered in the relaxed plan. This is further aggravated if helpful actions are used, as the action of picking the bomb up is never considered as helpful.

The first steps of the backwards generation are the following (assume that c1 and c2 are adjacent cells):

```
Goal: (holds-gold)

Step 1: (pick-gold c2) -> (holds-gold)
-New goal: (robot-at c2), (gold-at c2), (arm-empty)

Step 2: (move c1 c2) -> (robot-at c2)
-New goal: (robot-at c1), (clear c2), (arm-empty),
           (gold-at c2)

Step 3: (fire-laser c1 c2) -> (clear c2)
                        ;; deletes (gold-at c2)
        (detonate-bomb c1 c2) -> (clear c2), (arm-empty)
-New goal: (robot-at c1), (holds-bomb), (gold-at c2)
```

The original goal is $G_1 = \{$*(holds-gold)*$\}$. There is only one action that supports it with no other effect, *(pick-gold c2)*, so the proposition it satisfied is substituted with its preconditions, generating a new set of goals, $G_2 = \{$*(robot-at c2), (gold-at c2), (arm-empty)*$\}$. After this step, there are two sets of goal propositions: the original one, G_1 and the new one, G_2. In the second step, G_2 is satisfied in an earlier step when expanding the relaxed plan graph. Both *(robot-at c2)* and *(arm-empty)* can be legally supported (*(gold-at c2)* is already true in the initial state). As it was explained before, ties among actions that legally support goal propositions are broken choosing those that support propositions first satisfied at later levels. Because of this, the action *(move c1 c2)* is the one chosen to generate an additional intermediate set of goals, G_3 ={*(robot-at c1), (clear c2), (arm-empty), (gold-at c2)*}.

In the third step, G_3 can be supported by several actions again. In particular, to clear the way towards the goal the robot can either fire the laser or use a bomb. Nevertheless, since delete effects are taken into account in the backwards goal generation (*consistency* is a requisite of legal support), *(fire-laser c1 c2)* appears as illegal and is discarded. Thus, *(detonate-bomb c1 c2)* is chosen instead. Using *(detonate-bomb c1 c2)* generates G_4 ={*(robot-at c1), (holds-bomb), (gold-at c2)*} as a new BGG, which captures the hardest difficulty in the domain and allows easily solving the problem - if G_4 was given to a heuristic search planner using the FF heuristic instead of the original goal, the instance would be most likely trivial, as the proposition *(holds-bomb)* strongly bias the generation of the relaxed plan and feeds the forward search with the correct helpful actions. Figure 5.5 illustrates schematically this sequence.

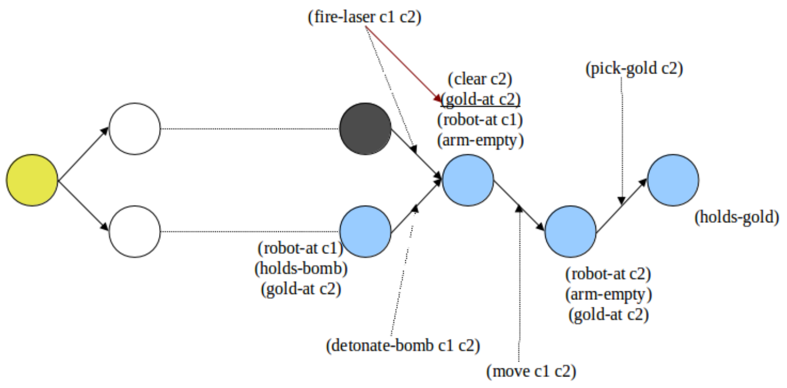


Figure 5.5: Example of how BGGs are generated in *Gold-Miner*.

5.5 Experimentation

Both the FF heuristic and the technique presented in this work (which we called BGG, Backwards Generated Goals) have been implemented on top of JavaFF (Coles et al., 2008). Because of the characteristics of BGG, it has been implemented as a forward state search algorithm with a dual queue (Röger and Helmert, 2010) as in Fast Downward. One of the queues uses the BGG heuristic, while the other uses the regular FF heuristic. The reason behind this is that, when computing the BGG heuristic, extracting a relaxed plan from the original goal takes little additional effort – the reachability analysis is already done and the best supporters are already computed for most propositions – and may be helpful for the cases in which the BGG heuristic is strongly guided towards an unreachable intermediate goal.

Greedy best-first search has been used as the search algorithm. Greedy best-first search expands in every step the most promising node determined by the function *f(n)* = *h(n)* in which *h(n)* is the heuristic function of the node. The use of intermediate goals requires some modifications, though. For every evaluated node, an intermediate goal is obtained. States with bad heuristic values are less likely to be on the path of a possible solution plan. Hence, they may yield intermediate goals that may mislead the search. To avoid this, intermediate goals are associated with the state along with its heuristic value in the open list instead of being added to the goal list. When a state is expanded then its intermediate goal is added to the list. This way, all the goals are generated from the best state at every iteration.

Helpful actions for the BGG heuristic are the union of the set of helpful actions obtained using the BGG relaxed plan and the regular relaxed plan from the original goal. Helpful action pruning has been enabled, but no restarts with the full set of successors have been done for the cases in which the planner was not able to find a solution with helpful action pruning under the time limit. This is done in order to analyze the impact of the extra helpful actions provided by the more informed relaxed plans from intermediate goals.

The focus of the experimentation is coverage (percentage of solved problems out of the total number of problems) of the proposed technique. Consequently, we have avoided some benchmark domains in which state-of-the-art planners are able to solve most problems in little time. In particular, the selected domains were *Gold-Miner*, *Matching-BW*, *N-Puzzle* and *Sokoban* from the learning track of IPC-6 and *Storage*, *Peg-Solitaire*, *Scananalyzer* and *Transportation* from the deterministic track of the same competition. For the sake of completeness, two other older domains were chosen as well, the well-known *Driverlog* domain from IPC-3 and the *Pipesworld* domain from IPC-4.

Experiments have been done on a Dual Core Intel Pentium D at 3.40 Ghz running under Linux. The maximum available memory for the planner was set to 1GB, and the time limit was 1800 seconds. The heuristic value of BGG is the distance to the closest intermediate goal without adding the distance from that goal to the original one. No parameter setting is necessary. Table 5.1 shows the results for these domains in terms of percentage of solved problems. Geometric mean and median of the ratio between states evaluated by BGG and FF are also displayed, with values above one meaning that BGG evaluates fewer nodes.

Overall, the coverage improves for BGG, but there is a notable trend: the harder a domain is, the greater the difference is between both techniques. Not taking into account Gold-Miner, which has a very hard constraint close to the goal and hence is the ideal case for BGG, domains in which BGG performs better have dead-ends and strong order interactions (Matching-BW, Sokoban) whereas in those without them there is no improvement. Regarding the number of evaluated nodes, BGG shows a similar behavior. Also, as the size of the problem increases, BGG tends to expand fewer nodes compared to the regular FF heuristic in spite of having a higher branching factor because of the additional helpful actions. This means that BGG is consistently more informed on the long run due to the increasing number

Domain	FF	BGG	Mean-S	Median-S
Driverlog	70	**80**	0.89	0.84
Gold-Miner	50	**100**	5.59	4.27
Matching-BW	3	**33**	9.72	9.72
N-Puzzle	10	**13**	4.13	4.25
Peg-Solitaire	**97**	80	1.67	1.71
Pipesworld	**28**	22	0.94	0.62
Scananalyzer	33	**67**	1.47	1.09
Sokoban	13	**17**	5.52	7.22
Storage	53	**60**	1.91	0.49
Transportation	63	**70**	0.7	0.68
Average	42	**57.5**	3.25	3.09

Table 5.1: Comparison between the FF heuristic and the BGG heuristic in terms of coverage (number of solved problems) and number of evaluated nodes (geometric mean and median of the ratio of evaluated states by FF to evaluated states by FF). Instances that were solved by both approaches expanding fewer than 100 nodes have not been considered.

of intermediate goals.

To better understand the differences between both approaches, some features of the domains that affect the performance of BGG can be analyzed. *Matching-BW* has numerous dead-ends which can be pruned earlier thanks to the constraints on the legality of the supporting actions; besides, towers of blocks impose order restrictions that can be found more easily using regression. *Sokoban* is characterized by having tunnels, which are implicitly exploited by the backwards goal generation, leading to a significant decrease in the depth of the search. *Storage* and *Scananalyzer* have similar characteristics to *Matching-BW*, but in *Storage* the search does not benefit from BGG as much as expected, probably because finding the correct goal ordering needs long sequences of actions in regression. *N-Puzzle* has no particularities that BGG may benefit from, and so coverage is not improved, but for those problems solved by both approaches BGG expands fewer nodes. *Pipesworld*, *Driverlog* and *Transportation* are domains in which the FF heuristic is relatively well informed, and hence the margin of improvement for BGG is small. *Peg-Solitaire* seems to be a similar case to *N-Puzzle*, although in this domain coverage decreases significantly for BGG.

Regarding quality, in those domains in which BGG improves the number of evaluated nodes it tends to improve plan quality as well, as seen in Table 5.2. In domains in which the number of evaluated nodes is similar, plan quality is worse for BGG, although on average BGG still finds slightly better plans. Analyzing the solution plans, it can be observed that the sequence of actions traced back from the reached intermediate goal tends to be of better quality than the part derived from the nodes expanded in the forward search. Still, experimentation has been done without adding the distance from the reached goal to the original one to the heuristic value of the evaluated nodes, which may prove useful when aiming for better quality plans.

Domain	Mean-Q	Median-Q	Ratio-BGG
Driverlog	1.08	1.08	0.17
Gold-Miner	2.35	2.76	0.27
Matching-BW	1.26	1.26	0.24
N-Puzzle	1.25	1.25	0.01
Peg-Solitaire	0.98	1	0.32
Pipesworld	1.04	1.04	0.14
Scananalyzer	0.94	0.85	0.25
Sokoban	1.09	1.11	0.19
Storage	0.93	0.93	0.21
Transportation	0.99	1	0.04
Average	1.19	1.24	0.18

Table 5.2: Comparison between the FF heuristic and the BGG heuristic in terms of quality. Again, geometric mean and median of the ratio between evaluated states has been used. Ratio-BGG is the percentage of the length of the solution part that belongs to the sequence of actions traced back from the reached intermediate goal.

Also, theoretically, the higher the average length of the part of the solution plan traced back from the reached intermediate goals is, the more relevant the backwards generation of goals is. Therefore, this can be considered a measure of how appropriate BGG is for some domains. Table 5.2 shows that this holds for most domains with the exceptions of *N-Puzzle* and *Transportation*. On average, around a fifth of the plan is retrieved from the reached intermediate goal. This also explains why BGG expands fewer nodes: the search space that must be explored to find a solution is potentially much smaller due to the reduced depth.

In terms of time, the increasing number of intermediate goals makes the heuristic more expensive to compute. *A priori*, since the number of goals grows exponentially, so should do the time needed to compute the heuristic. In practice, however, the time needed increases linearly because intermediate goals tend to appear at earlier levels as the search progresses and those at later levels are not checked. In general, in those domains in which the number of expanded nodes decreases when using BGG, it tends to find the solution in a similar time than when using the FF heuristic; in domains in which BGG does not improve the number of nodes, planning time may increase up to an order of magnitude. Figure 5.6 shows the time needed to compute the heuristic as the number of intermediate goals increases in the eleventh problem of the *Pipesworld* domain. As it can be observed, the main tendency shows that time increases linearly and only slightly, going from around

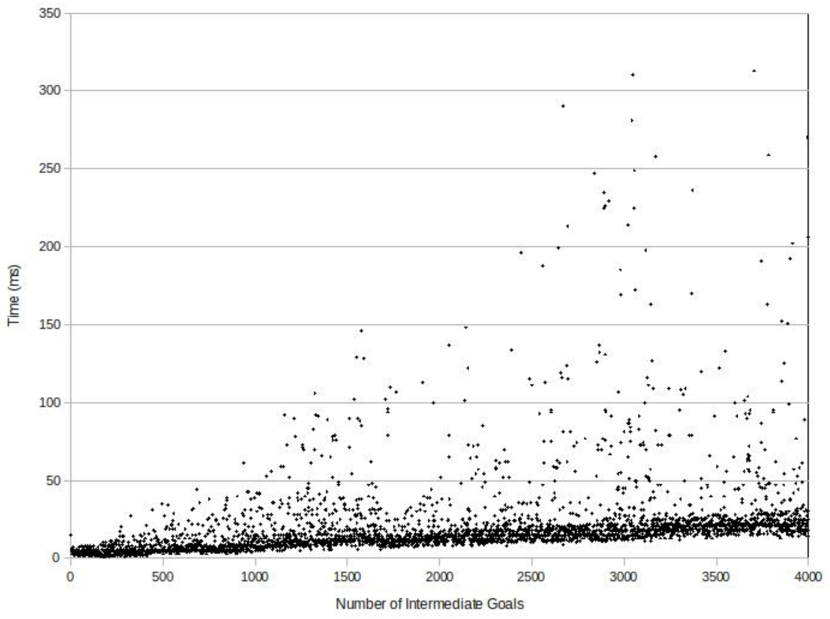


Figure 5.6: Time needed to compute the heuristic in milliseconds as the number of goals increase.

5 milliseconds at the beginning to around 25 milliseconds when 4000 intermediate goals have been generated. Still, there are two important factors that make BGG more costly in terms of time: first, the number and magnitude of outliers increase as intermediate nodes are generated; and second, heuristic computation accounts for most of the time spent by forward search heuristic planners, so even small increases in time in the heuristic computation have a great impact on the overall performance.

5.6 Related Work

As BGGs are generated in regression while using a forward search algorithm, a clearly related concept is bidirectional search. In fact, BGGs are partial states identical to those generated in regression; the difference is the generation method. Whereas BGGs are generated from a relaxed plan computed in progression, partial states are generated by a search algorithm and a specific heuristic in regression. The use of BGGs to enrich the computation of reachability heuristics in progression is akin to front-to-front heuristics, which estimate the distance from one search frontier to the other instead of from a given state to its correspondent goal.

From a procedural point of view, another recent work on low-conflict relaxed plans (Baier and Botea, 2009) shares important aspects with this approach. In their work, they take into account delete effects and mutexes in the backwards process done to retrieve relaxed plans to obtain more accurate estimations. Furthermore, they expand additional levels in the computation of the heuristic; the main difference with our approach is that in our work the additional expansion of levels is systematic (until a legal support is found) whereas in their case they require a preliminary estimation of the number of extra levels that may be useful to improve the heuristic.

5.7 Conclusions and Future Work

The main contribution of this work consists on understanding the relationship between the last actions of the relaxed plans obtained in the computation of the FF heuristic with potential solution plans, and how this idea can be used to improve performance. We have presented a new approach that combines forward state search and backwards goal generation with reachability heuristics and relaxed plans as the common core. BGG presents several advantages: it reduces the depth in both the relaxed graph used in the heuristic computation and the state space search; it detects constraints close to the goal, which also provides an advantage for the forward state search by generating more informed relaxed plans; and it uses additional information, such as delete effects and mutexes, to enrich the reachability analysis.

There are some disadvantages derived from the use of intermediate goals: the heuristic must take into account an increasing number of goals, which is exponential in the worst case (although this effect has not been found in the experiments); and the distance to an intermediate goal used as the heuristic value by the nodes in the open list becomes obsolete as new intermediate goals are generated. Summarizing the results of the experiments, it usually pays off to generate intermediate goals during search. Out of the ten domains used, it improves the coverage in all but two of them. Besides, it is competitive in terms of time and quality as well. It also seems to work better in domains traditionally hard for regular reachability heuristics, allowing planners to scale not only on size but also on the inherent complexity of the domains.

On the other hand, there are some techniques that intuitively seem to be well suited to the backwards generation of goals. Look-ahead techniques are a prominent example; the twist in this case is to combine actions from the last part of the relaxed plan to generate intermediate goals more than one step farther from the reached goal. Another technique that offers a great potential is the combination of this work with that of landmarks (Hoffmann et al., 2004).

It would also be interesting to analyze the potential links between backwards goal generation and bidirectional/perimeter search (Dillenburg and Nelson, 1994). As mentioned before, this approach can be seen as a sort of imbalanced bidirectional algorithm in which the backwards search uses the last actions of the relaxed plan as a guide and the forward search estimates the distance to the frontier of the backwards search instead of to the original goal.

Part IV

Rapidly-exploring Random Trees and Sampling in Planning

Chapter 6

Adapting RRTs for Automated Planning

Rapidly-exploring random trees (RRTs) are data structures and search algorithms designed to be used in continuous path planning problems. They are one of the most successful state-of-the-art techniques in motion planning, as they offer a great degree of flexibility and reliability. However, their use in other fields in which search is a commonly used approach has not been thoroughly analyzed. In this work we propose the use of RRTs as a search algorithm for automated planning. We analyze the advantages and disadvantages that this approach has over previously used search algorithms and the challenges of adapting RRTs for implicit and discrete search spaces.

6.1 Introduction

Currently most of the state-of-the-art planners are based on the heuristic forward search paradigm first employed by HSP (Bonet and Geffner, 2001). While this represented a huge leap in performance compared to previous approaches, this kind of planners also suffers from some shortcomings. In particular, certain characteristics of the search space of planning problems hinder their performance. Large plateaus for h values, local minima in the search space and areas in which the heuristic function is misleading represent the main challenges for these planners. Furthermore, most successful planners use techniques that increase the greediness of the search process, which often exacerbates this problem. A couple of examples of these approaches are pruning techniques like helpful actions introduced by FF (Hoffmann and Nebel, 2001) and greedy search algorithms like Enforced Hill Climbing (EHC) used by FF and greedy best-first search used by HSP and Fast Downward (Helmert, 2006).

Motion planning is an area closely related to automated planning. Problems in motion planning consist on finding a collision-free path that connects an initial configuration of geometric bodies to a final goal configuration. Some examples of motion planning problems are robotics (Goerzen et al., 2010), animated simulations (Christie et al., 2005), drug design (Cortés et al., 2005) and manufacturing (Da Xu et al., 2012) to name a few. A broad range of techniques have been used in this area, although the current trend is to use algorithms that randomly sample the search space due to their reliability, simplicity and consistent behavior. Probabilistic roadmaps (PRMs) (Kavraki et al., 1996) and Rapidly-exploring Random Trees (RRTs) (LaValle and Kuffner, 1999) are the most representative techniques based on this approach.

Algorithms based on random sampling have two main uses: multi-query path planning, in which several problems with different initial and goal configurations must be solved in the same search space, and single-query path planning, in which there is only a single problem to be solved for a given search space. In the case of single-query path planning, RRT-Connect (Kuffner and LaValle, 2000), a variation of the traditional RRTs used in multi-query path planning, is one of the most widely used algorithms. RRT-Connect builds a tree from the initial and the goal configurations by iteratively extending towards sampled points while trying to join the newly created branches with the goal or with a node belonging to the opposite tree. This keeps a nice balance between exploitation and exploration and is often more reliable than previous methods like potential fields, which tend to get stuck in local *minima*.

Single-query motion planning and satisficing planning have many points in common. However, bringing techniques from one area to the other is not straightforward. The main difference between the two areas are the defining characteristics of the search space. In motion planning, the original search space of these problems is an euclidean explicit continuous space, whereas in automated planning the search space is a multi-dimensional implicit discrete space. This has lead to both areas being developed without much interaction despite the potential benefits of an exchange of knowledge between the two communities.

In this work, we try to bridge the gap between the two areas by proposing the use of an RRT in automated planning. The motivation is that RRTs may be able to overcome some of the shortcomings that forward search planners have while keeping most of their good properties. The major contributions of this work are the following:

- We describe how to implement a search algorithm for domain-independent planning based on RRTs.
- We propose a general and efficient way of employing domain-independent reachability heuristics for their use as the distance estimator in the RRT.
- We present a method based on constraint satisfaction to sample implicit search spaces uniformly.
- We perform experimentation over a broad set of domains analyzing the impact of the different parameters that characterize the algorithm.

This chapter is organized as follows: first, some background and an analysis of previous works will be given; next, the advantages of using RRTs in automated planning will be presented as well as a description of how to overcome some problems regarding their implementation; later, some experimentation will be done to back up our claims; and, finally, some conclusions and future lines of research will be added.

6.2 Background

In this section we will give some necessary background about RRTs. This includes both the original definition of RRTs as a data structure and its subsequent evolution as a single-query search algorithm in motion planning.

6.2.1 Rapidly-exploring Random Trees

RRTs (LaValle and Kuffner, 1999) were proposed as both a sampling algorithm and a data structure designed to allow fast searches in high-dimensional spaces in motion planning. RRTs are progressively built towards unexplored regions of the space from an initial configuration. Configurations describe the position, orientation and velocity of the movable objects in motion planning, and are equivalent to states in other search applications.

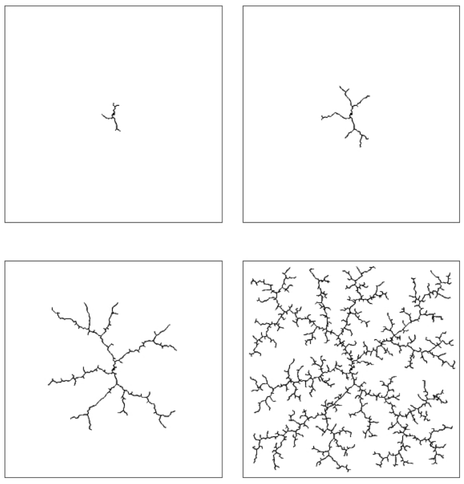


Figure 6.1: Progressive construction of an RRT.

Figure 6.1 shows how an RRT is grown from the initial configuration. At the start, the algorithm creates a tree containing the initial configuration. At every step, a random q_{rand} configuration is chosen from all the configuration space and for that configuration the nearest configuration already in the tree q_{near} is computed. For this a definition of distance is required (in motion planning the euclidean distance is usually chosen as the distance measure). When the nearest configuration is found, a local planner tries to join q_{near} with q_{rand} with a limit distance ϵ. If q_{rand} was reached, it is added to the tree and connected with an edge to q_{near}. If q_{rand} was not reached, then the configuration q_{new} obtained at the end of the local search is added to the tree in the same way as long as there was no collision with an obstacle during search. In the search literature, the term *local search* refers to search algorithms that do not keep track of all the states that they have visited. The most representative algorithm of this kind is Hill Climbing, although many others exist. Here, though, whenever we use the term *local search* we mean the process of solving the subproblem needed to create a new branch of the tree. This operation is called the *Extend* step, illustrated in Figure 6.2. This process is repeated until some criteria is met, like a limit on the size of the tree. Algorithm 3 gives an outline of the process.

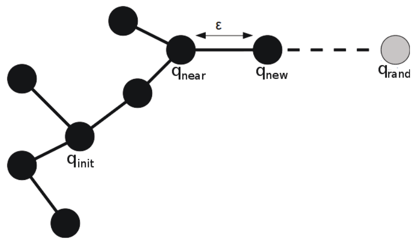


Figure 6.2: Extend phase of an RRT.

Algorithm 3: Description of the building process of an RRT.

```
Data: Search space $S$, initial configuration $q_{init}$, limit $\epsilon$, ending criteria $end$
Result: RRT $tree$
begin
    $tree \longleftarrow q_{init}$
    while $\neg$ $end$ do
        $q_{rand} \longleftarrow$ sampleSpace($S$)
        $q_{near} \longleftarrow$ nearest($tree, q_{rand}, S$)
        $q_{new} \longleftarrow$ join($q_{near}, q_{rand}, \epsilon, S$)
        if reachable($q_{new}$) then
            addConfiguration($tree, q_{near}, q_{new}$)
    return $tree$
end
```

Once the RRT has been built, multiples queries can be issued. For each query, the nearest configurations (node) of the tree to both the initial and the goal configurations of the query are found. Then, the initial and final configurations are joined to the tree to those nearest configurations using the local planner and a path is retrieved by tracing back edges through the tree structure.

The key advantage of RRTs is that they are intrinsically biased towards regions with a low density of configurations in their building process. This can be explained by looking at the Voronoi diagram at every step of the building process. The Voronoi diagram is conformed by Voronoi regions; Voronoi regions associated to a given node q of the tree are areas such that every point in the area is closer to q than to any other node q' of the RRT. The Voronoi region of a given node is larger when the area around that node has not been explored. This way, the probability of a configuration being sampled in an unexplored region is higher as larger Voronoi regions will be more likely to contain the sampled configuration (Aurenhammer, 1991). This has the advantage of naturally guiding the tree by extending nodes at the edge of unexplored regions with a higher probability while just performing uniform sampling. Besides, the characteristics of the Voronoi diagram are an indicative of the adequateness of the tree. For example, a tree whose Voronoi diagram is formed by regions of similar size covers uniformly the search space, whereas large disparities in the size of the regions mean that the tree may have left big areas of the search space unexplored. Apart from this, another notable characteristic is that RRTs are probabilistically complete, as they will cover the whole search space if the number of sampled configurations tends to infinity. Figure 6.3 shows the Voronoi diagrams of the RRTs previously shown in Figure 6.1[1].

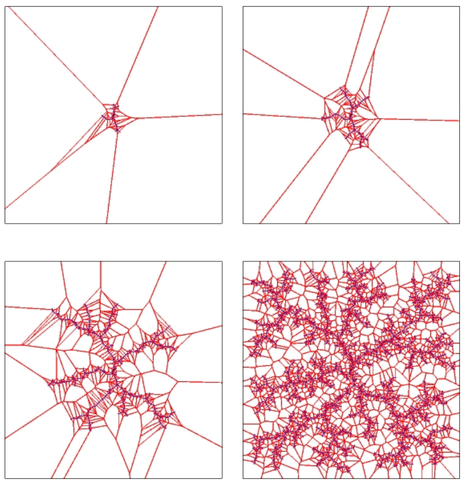


Figure 6.3: Voronoi Diagram of an RRT.

6.2.2 RRT-Connect

After corroborating how successful RRTs were for multi-query motion planning problems, researchers in motion planning realized that using multi-query RRTs was often more efficient and robust than using specific single-query motion planning algorithms even for a single query. Motivated by this fact and aiming to develop a more suitable RRT-like algorithm for the single-query case, a variation for single-query problems called RRT-Connect was proposed (Kuffner and LaValle, 2000).

[1]Both figures were taken from LaValle's slides about RRTs

The modifications introduced were the following:

- Two trees are grown at the same time by alternatively expanding them. The initial configuration of the trees are the initial and the goal configuration respectively.
- The trees are expanded not only towards randomly sampled configurations, but also towards the nearest node of the opposite tree with a probability p. Hence, with a probability p the closest distance among the $m \times n$ distances between nodes from both trees is found and the node of the expanding tree is expanded towards the node of the non-expanding tree. With a probability $1 - p$ a random configuration is sampled and the corresponding trees are expanded as usual.
- The *Expand* phase is repeated several times until an obstacle is found. The resulting nodes from the local searches limited by ϵ are added to the tree. This is called the *Connect* phase.

Growing the trees from the initial and the goal configurations and, at times, towards the opposite tree gives the algorithm its characteristic single-query behavior. The *Connect* phase was added after empirically testing that the additional greediness that it introduced improved the performance in many cases. A common variation is also trying to extend the tree towards the opposite tree after every q_{new} is added when sampling randomly, extending from that q_{new} configuration towards the opposite tree. This helps in cases in which both trees are stuck in regions of the search space that are close as per the distance measure, but in which local searches consistently fail due to obstacles.

6.3 Previous Work

Although RRTs have not been frequently used in areas other than motion planning, there is previous work in which they have been employed for problems relevant to automated planning. In particular, an adaptation of RRTs for discrete search spaces and a planning search algorithm similar to RRT-Connect have been proposed.

6.3.1 RRTs in Discrete Search Spaces

Although RRTs were designed for continuous search spaces, researchers from other areas proposed their implementation for search problems with discrete search spaces (Morgan and Branicky, 2004). The main motivation of that work was adapting the RRTs to grid worlds and similar classical search problems.

Its authors analyzed the main challenges of adapting RRTs, in particular the need of defining an alternative measure of distance to find the nearest node of the tree to a sampled state and the issues related to adapting the local planner that must substitute the *Expand* phase of the traditional RRTs. The proposed alternative to the widely used euclidean distance was an "ad-hoc" heuristic estimation of the cost of reaching the sampled state from a given node of the tree. As for the local planner, the limit ϵ that was used to limit the reach of the *Expand* phase was substituted by a limit on the number of nodes expanded by the local planner. In this case, once the limit ϵ was reached the node in the local search with the best heuristic estimate towards the sampled space was chosen, and either only that node or all the nodes on the path leading to it were added to the tree.

While the approach was successful for the proposed problems, there are two main problems that make it impossible to adapt it to automated planning in a straightforward way. First, the search spaces in the experimentation they performed are explicit, whereas in automated planning the search space is implicit. This adds an additional complexity to the sampling process that must be dealt with in some way. Second, the heuristics for both the distance estimation and the local planners were designed individually for every particular problem and thus are not useful in the more general case of automated planning.

6.3.2 RRT-Plan

Directly related to automated planning, the planner RRT-Plan (Burfoot et al., 2006) was proposed as a stochastic planning algorithm inspired by RRTs. In this case, the EHC search phase of FF (Hoffmann and Nebel, 2001), a deterministic propositional planner, was used as the local planner. The building process of the RRT was similar to the one proposed for discrete search spaces; that is, to impose a limit on the number of nodes as ϵ and add the expanded node closest to the sampled state to the tree. In this case, though, the tree was built only from the initial state due to the difficulty of performing regression in automated planning.

The key aspects in this work are two: the computation of the distance necessary to find the nearest node to the sampled or the goal state, and sampling in an implicit search space. In RRTs one of the most critical points is the computation of the nearest node in every *Expand* step, which may become prohibitively expensive as the size of the tree grows with the search. The most frequently used distance estimations in automated planning are the heuristics based on the reachability analysis in the relaxed problem employed by forward search planners, like the h^{add} heuristic used by HSP (Bonet and Geffner, 2001) or the relaxed plan heuristic introduced by FF (Hoffmann and Nebel, 2001). The problem with these heuristics is that, although computable in polynomial time, they are usually still relatively expensive to compute, to the point that they usually constitute the bottleneck in satisficing planning. To avoid recomputing the reachability analysis from every node in the tree, every time a new local search towards a state is done, the authors propose caching the cost of achieving every goal proposition from a node whenever that node is added to the tree. This way, by adding the costs of the propositions that form the sampled state, h^{add} can be obtained without needing to perform a reachability analysis.

Regarding sampling, RRT-Plan does not sample the search state uniformly. Instead, it chooses a subset of propositions $s \subseteq S$ from the goal set such that $s \subseteq G$ and uses s as q_{rand}. This is due to the fact that, although sampling a state by choosing random propositions in automated planning is trivial, determining whether a given sampled state belongs to the search space is PSPACE-complete, as it is as hard, in terms of computational complexity, as solving the original problem itself. This problem is avoided by the sampling technique of RRT-Plan in the sense that, if the problem is solvable, G must be reachable. Hence, any of its possible subsets is also reachable. In addition, RRT-Plan performs goal locking; i.e., when a goal proposition p that was part of a given sampled state $s \subseteq G \mid p \in s$ is achieved, any subsequent searches from the added q_{new} node and its children nodes are not allowed to delete p.

Whereas RRT-Plan effectively addresses the problem of sampling states in implicit search spaces, this kind of sampling limits most of the advantages RRTs have to offer. By choosing subsets of the goal set instead of sampling uniformly the search space, the RRT does not tend to expand towards unexplored regions. Thus, it loses the implicit balance between exploration and exploitation during search that characterizes them. In fact, by choosing this method, RRT-Plan actually benefits from random guesses over the order of the goals instead of exploiting the characteristics of RRTs. As a side note, this could actually be seen as a method similar to the goal agenda (Koehler and Hoffmann, 2000), albeit with random selection of subsets and the possibility in this case to recover from wrong orderings.

6.4 Advantages of RRTs in Automated Planning

As mentioned in the introduction, during the last years there has been a big improvement in performance in the area of propositional planning. The most representative approach among those that contributed to this improvement is the use of forward heuristic search algorithms along with reachability heuristics and other associated techniques. However, heuristic search planners suffer from several problems that detract from their performance. These problems are related mainly to the characteristics of the search space that most common planning domains have. Search spaces in automated planning tend to have big plateaus in terms of the h value. The high number of transpositions and the high branching factor that are characteristic of many domains aggravate this fact. Heuristic search planners that use best-first search algorithms are particularly affected by this, as they consider total orders when generating new nodes and are mostly unable to detect symmetries and transpositions during search. It has been shown that techniques that increase the greediness of the search algorithm, like helpful actions from FF and look-ahead states from YAHSP (Vidal, 2004b), tend to partially alleviate these problems. However, even though reachability heuristics have proved to be relatively reliable for the most part, in some cases they can also be quite misguided. This increased greediness can be disadvantageous at times.

Figure 6.4 shows a typical example of a best-first search algorithm getting stuck in an *h plateau* due to inaccuracies in the heuristic. In this example, the euclidean distance used as heuristic ignores the obstacles. Because of this, the search advances forward until the obstacle is found. Hence, the search algorithm must explore the whole striped area before it can continue advancing towards the goal. This highlights the imbalance between exploitation and exploration these approaches have. This problem has been previously studied, and several methods that tried to minimize its negative impact on search have been proposed (Röger and Helmert, 2010; Linares López and Borrajo, 2010). However, this imbalance still remains as one of the main shortcomings of best-first search algorithms. To partially address this issue, we consider expanding nodes towards randomly sampled states so a more diverse exploration of the search space is done. In this example, a bias that would make the search advance towards q_{rand} could avoid the basin flooding phenomenon that greedier approaches suffer from.

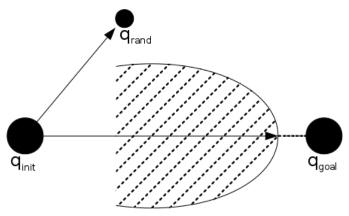


Figure 6.4: Simple example of a best-first search algorithm greedily exploring an *h plateau* due to the heuristic ignoring the obstacles. Advancing towards some randomly sampled state like q_{rand} can alleviate this problem.

RRTs incrementally grow a tree towards both randomly sampled states and the goal. Therefore, they are less likely to suffer from the same problem as best-first search algorithms. The main advantages that they have over other algorithms in automated planning are the following:

- They keep a better balance between exploration and exploitation during search.
- Local searches minimize exploring plateaus, as the maximum size of the search is bounded.
- They use considerably less memory, as only a relatively sparse tree must be kept in memory.
- They can employ a broad range of techniques during local searches.

All in all, the alternation of random sampling and search towards the goal that single-query RRTs have is their most characterizing aspect. Thanks to this, they do not commit to a specific area of the search space and hence they tend to recover better than best-first search from falling into local *minima*. In terms of memory, the worst case is the same for best-first search algorithms and RRTs. However, RRTs must keep in memory only the tree and the nodes from the current local search.

Trees are typically much sparser than the area explored by best-first algorithms, which makes them much more memory efficient on average. Memory is usually not a problem in satisficing planning because of the time needed for the heuristic computation, although in some instances it can be an important limiting factor.

6.5 Implementing the RRT for Automated Planning

Due to the differences in the search space, adapting RRTs from motion planning to automated planning is not trivial. In this work we propose an implementation partially based on RRTs for discrete search spaces and RRT-Plan with some changes critical to their performance.

6.5.1 Sampling

The main reason why RRTs have not been considered for automated planning is the difficulty of properly sampling the search space. The difficulty arises from the fact that choosing propositions from S at random may lead to generating spurious states. RRT-Plan circumvented this by substituting uniform sampling with subsets of goal propositions. However, this negates some of the advantages that RRTs have, as explained before.

Checking whether a state is spurious or not is as hard as solving the problem itself, so an approximation must be used instead. Here we propose the use of state invariants as constraints to reduce the chances of obtaining a spurious state when uniformly sampling the search space. In particular, we propose the use of mutexes (already employed by evolutionary planners like D_AE_X (Bibai et al., 2010), which decomposes the problem using sampling techniques) and *"exactly-1"* invariant groups.

Selecting Propositions

Sampling a state using state invariants as constraints is analogous to solving a Constraint Satisfaction Problem (CSP). A CSP is defined as a triple *CSP=(V,D,C)*, where *V* are the variables of the problem, *D* are the domains of the variables in *D* and *C* are the constraints of the problem. In this CSP the *"exactly-1"* invariant groups are the variables in *V*, the propositions of the *"exactly-1"* invariant groups are the domain *D* of the variables in *V* and the binary mutexes of the problem are the constraints of *C*. The objective is to choose a proposition $p \in S$ from every *"exactly-1"* invariant group I_1 such that it is not mutex with any other chosen proposition $p' \in S$. This ensures that the complete sampled state $s \in S$ satisfies all *"exactly-1"* invariant groups and does not violate any binary mutex. This is in fact conceptually close to the disambiguation process presented in Section 3.4.1.

Solving a CSP is NP-complete. Actually, for some planning instances solving the CSP that represents the sampling process may be on average very time consuming if it is done naively. In our implementation we use forward checking (Haralick and Elliott, 1980) to improve the performance of the backtracking procedure needed for solving the CSP. The order of the variables (the order in which the *"exactly-1"* invariant groups are selected to be satisfied) is static, although it may vary between different sampling processes. *"exactly-1"* invariant groups with the highest cardinality are chosen first, with ties broken randomly every time a new state is sampled. Ordering of values of variables is chosen at random. This aims to reproduce the behavior of the degree (most constraining variable) heuristic (Brélaz, 1979) while trying to obtain sampled states as diverse as possible.

Ensuring the Reachability of Goals

Even after using state invariants, it may happen that the goal is not reachable from the sampled state. For example, a sampled state in the *Sokoban* domain may contain a configuration of blocks such that some block cannot be moved anymore. While this sampled state may not violate any state invariant, unless the unmovable blocks are at a goal location the sampled state is a dead end, since the original goal is not reachable. To address this problem, a regular reachability analysis can be done from the sampled state. If some proposition $p \in G$ is unreachable, then the sampled state can be safely discarded. This is again an incomplete method, but in cases such as the aforementioned one it is useful to detect spurious states.

6.5.2 Distance Estimation

One of the most expensive steps in an RRT is finding the closest node to a sampled state. Besides, the usual distance estimation in automated planning, the heuristics derived from a reachability analysis, are also computationally costly. RRT-Plan solved this by caching the cost of achieving a goal proposition from every node of the tree and using that information to compute h^{add}, just like HSPr does (Bonet and Geffner, 2001) when searching backwards. Despite being an efficient solution, this shares the same problem as HSPr: only h^{add} can be computed using that information. h^{add} tends to greatly overestimate the cost of achieving the goal set and other heuristics of the same kind, like the FF heuristic, are on average more accurate (Fuentetaja et al., 2009). Therefore, in our implementation, *best supporters*, that is, actions that first achieve a given proposition in the reachability analysis, are cached, as defined in Section 3.5.1. This allows to compute not only h^{add} but also other heuristics like the FF heuristic (by tracing back the relaxed plan using the cached best supporters). The time of computing the heuristic once the best supporters are known is usually very small compared to the time needed to perform the reachability analysis - linear in the size of the relaxed plan -, so this approach allows to get more accurate (or diverse) heuristic estimates without incurring in a significant overhead.

6.5.3 Tree Construction

RRTs can be built in several ways. The combination of the *Extend* and *Connect* phases, the possibility of greedily advancing towards the goal with a probability $1-p$ instead of sampling with a probability p, the way new nodes are added (only the closest node to the sampled state or all the nodes on the path to that state),... allow for a broad range of different options. In this work, we have chosen to build the tree in the following way:

- the tree is built from the initial state I;
- every node in the tree contains a state, a link to its parent, a plan that leads from its parent to the state, and the cached best supporters for every proposition $q \in S$ so h^{FF} can be computed efficiently;
- ϵ limits the number of expanded nodes in every local search;
- there is a probability p of advancing towards a sampled state and a probability $1-p$ of advancing towards the goal from the closest node to the original goal G. It may happen that the closest node to G was already expanded towards G in an earlier iteration and the new generated node q_{new} from that expansion is farther from the goal than G; that is, $h^{FF}(q_{new}) > h^{FF}(q_{near})$. Since planners are for the most part deterministic, it does not make sense to repeat the search - it would lead to the same q_{new} -, so in fact the node selected with a probability $1-p$ is the closest node *among those that have never served as the origin of a local search towards the goal before*;
- a single node is added to the tree after every local search, not all the nodes along the solution path;
- when performing a local search, if a solution for the subproblem was not found, the last expanded node is added to the tree (be it when expanding towards a sampled state or the original goal G itself);
- after adding a new node q_{new} from the local search towards a sampled state, a new local search from q_{new} to q_{goal} is performed.

No *Connect* phase is performed. This is because the *Connect* phase is probably counter-productive if it is done towards sampled states - sampled states may be completely irrelevant to the solution and the main benefit obtained from them is the additional bias towards exploration anyway - and partially overlaps with the expansions towards the goal with a probability $1-p$.

This configuration is the basis of the planner presented in this work; we called this planner Randomly-exploring Planning Tree (RPT). Algorithm 4 describes the whole process.

Algorithm 4: Search process of RPT.

```
Data: Search space $S$, limit $\epsilon$, initial state $q_{new}$, goal $q_{goal}$
Result: Plan $solution$
begin
    $tree \longleftarrow q_{init}$
    while $\neg$ goalReached() do
        if $p$ <random() then
            $q_{rand} \longleftarrow$ sampleSpace(S)
            $q_{near} \longleftarrow$ nearest(tree,$q_{rand}$,S)
            $q_{new} \longleftarrow$ join($q_{near}, q_{rand}, \epsilon$,S)
            addNode(tree,$q_{near}, q_{new}$)
            $q_{near_{goal}} \longleftarrow q_{new}$
        else
            $q_{near_{goal}} \longleftarrow$ nearest(tree,$q_{goal}$,S)
        $q_{new_{goal}} \longleftarrow$ join($q_{near_{goal}}, q_{goal}, \epsilon$,S)
        addNode(tree,$q_{near_{goal}}, q_{new_{goal}}$)
    solution $\longleftarrow$ traceBack(tree,$q_{goal}$)
    return solution
end
```

6.5.4 Choice of the Local Planner

The choice of the planner used in the local search is subject to some restrictions. First, after every *Extend* phase a new node to the tree is added even if a solution for the subproblem could not be found. This means that the local planner must be able to return an executable plan also when no solution was found, which rules out some planning paradigms like partial-order planners (Younes and Simmons, 2003) and SAT-based planners (Rintanen, 2010). Second, the tree is built forward, so the local planner must return a forward-executable plan. Again, backward search planners like HSPr (Bonet and Geffner, 2001) and FDr cannot be used for this reason. Another important point is the preprocessing time. Since multiple local searches may be done, it is desirable that the time spent by the local planner prior to search is as short as possible. For example, the use of heuristics that require a relatively long preprocessing time and depend on either the initial state or the goals, like Pattern Databases (Culberson and Schaeffer, 1998), are discouraged.

In this work, the Fast Downward planning system (Helmert, 2006) was used as the local planner. It was configured to use greedy best-first search with lazy evaluation as its search algorithm. The heuristic is the FF heuristic (Hoffmann and Nebel, 2001). Preferred operators obtained from the FF heuristic were enabled.

6.6 Experimentation

In this section, we test the proposed approach against other state-of-the-art planners. The maximum available memory was set to 6GB and the time limit was 1800 seconds, as in the last International Planning Competition (IPC-2011). The selected domains were all the domains from the deterministic track of the past IPCs. In the cases in which a domain appeared in more than one competition, we selected the instances from the hardest set. The criteria are the same as the ones used by Rintanen (Rintanen, 2010). Additionally, we used the test problems from the learning track of the 2008 and 2011 IPCs for those domains that were not used in the deterministic track. The exception is the *Sokoban* domain; the definition of the domain and the structure of the problems are significantly different from the ones of the deterministic track, so we employed both versions. We call the version from the learning track *Sokoban-l*. In this section, we will focus on the number of problems solved (also known as coverage), so all actions will be treated as if they had unary cost.

RPT was implemented on top of Fast Downward (Helmert, 2006). Since RRTs are stochastic algorithms, all the experiments are done using the same seed for the pseudorandom number generator so results would be reproducible. In particular, we use the default seed '1' for the *rand* function of the standard C++ library. The computation of h^2 was implemented in Fast Downward and the mutexes obtained from the computation of h^2 forward and backward. *"exactly-1"* invariant groups were obtained from the monotonicity analysis done by the translator of Fast Downward. To further exploit the state invariants, spurious actions were pruned by disambiguating their preconditions, as in Section 3.4.1. We set a limit of 300 seconds for the h^2 computation and the disambiguation of actions.

The planners we compare against are the local planner itself, that we call FD, and the first phase of the LAMA planner (Richter and Westphal, 2010), the winner of the IPCs held in 2008 and 2011. We used the last revision from Fast Downward's public repository for both planners.[2] In fact, LAMA is the same as FD, but also uses an additional landmark counting heuristic combined with the regular FF heuristic. Preferred operators are obtained from the landmark heuristic too and the final set of preferred operators is the union of the sets of preferred operators obtained from both heuristics. Both heuristics are combined in an alternation queue. This queue expands the best node as per the correspondent heuristic alternatively at every node expansion.

There are two critical parameters that affect the behavior of RPT in our implementation: the limit on the number of expanded nodes in the local search ϵ and the probability p of expanding towards a sampled state instead of towards the original goal G. In the experimentation we tried six different combinations resulting from combinations of ϵ and p. The values used for ϵ were $\epsilon = 10000, 100000$. The values used for p were $p = 0.2, 0.5, 0.8$. An additional configuration with an ar-

[2]As of December 2013, revision 3288.

tificially low ϵ of 1000 nodes and $p = 0.5$ was also used to test the performance of RPT when using very small local searches. By experimenting with different versions we aim to understand which are the factors that can have an impact on the performance of the algorithm.

6.6.1 Coverage

Table 6.1 shows the coverage of all the evaluated planners. It also shows the number of tree edges that the solution with the highest number of tree edges in that domain has for all the RPT configurations (between brackets). Results show that RPT is overall better than the base planner FD and competitive with the state-of-the-art planner LAMA. Total coverage favors all the tested configurations of RPT over both FD and LAMA except for the version with $\epsilon = 1000$.

A more detailed comparison on a per domain basis shows that performance depends in a significant number of cases on the domain. For instance, RPT is notably better in *Airport*, *Floortile*, *Sokoban-l* and *Storage*, whereas FD and LAMA are better in *Barman* and *N-puzzle*. The domains in which RPT performs better are often those in which there are dead ends undetectable by reachability heuristics, which cause FD and LAMA to explore big sterile plateaus. Note that *Floortile*, *Sokoban-l* and *Storage* have a similar structure, as achieving the closest goal proposition first often leads to a dead end; in this case the additional exploration induced by the sampling method and the limit on the number of expanded nodes by the local search ϵ prove to be very beneficial.

Some domains in which RPT is outperformed by LAMA are domains in which state invariants do not suffice to avoid spurious states. *Barman* is the most notable example, because the current definition allows some unintended things to happen (such as a glass containing a drink and being clean at the same time), which reduces the number of mutexes found by invariant computation methods. Another very relevant case is *Visitall*, in which FD is able to solve only 3 problems, LAMA solves the whole set of problems and RPT is somewhere in the middle. The difference between FD and LAMA is explained by the heuristics they use: *Visitall* is a domain designed to induce plateaus when the FF heuristic is used, but it is otherwise easily solved with heuristics that employ additive schemes such as the goal-counting and the landmark-counting heuristics. However, this does not explain why RPT behaves like it does. An important fact is that sampling in *Visitall* is very problematic, as it is possible to sample a state in which a visited cell can be surrounded by not-visited cells, which is unreachable from I. However, RPT still solves more problems than FD, which shows that the possibility of recovering from exploring plateaus that RPT has compensates the added difficulty of sampling the search space adequately.

In a few domains RPT is also significantly helped by pruning spurious actions. *Floortile*, *Matching-bw* and *Tidybot* are such domains. In a modified version of FD that performs the same preprocessing as RPT, pruning spurious actions reduces the difference in coverage in these domains by a large margin even if state invariants

Planners:	FD	LAMA	10k_0.2	10k_0.5	10k_0.8	100k_0.2	100k_0.5	100k_0.8	1k_0.5
Airport(50)	35	32	**46** [5]	**46** [7]	**46** [4]	40 [3]	44 [3]	45 [3]	**46** [7]
Barman(20)	**20**	**20**	3 [2]	5 [2]	2 [2]	11 [11]	10 [21]	9 [13]	2 [10]
Blocks(35)	**35**	**35**	**35** [1]	**35** [1]	**35** [1]	**35** [1]	**35** [1]	**35** [1]	**35** [5]
Depot(22)	18	**21**	19 [28]	20 [32]	20 [16]	18 [9]	19 [6]	19 [7]	19 [16]
Driverlog(20)	**20**	**20**	**20** [5]	**20** [13]	**20** [10]	**20** [5]	**20** [7]	**20** [6]	18 [8]
Elevators(20)	19	**20**	**20** [10]	**20** [9]	**20** [4]	**20** [5]	**20** [5]	**20** [4]	**20** [71]
Floortile(20)	7	6	**20** [22]	**20** [12]	**20** [4]	**20** [2]	**20** [2]	**20** [1]	**20** [53]
Freecell(80)	79	79	**80** [5]	**80** [2]	**80** [2]	**80** [2]	**80** [2]	**80** [3]	**80** [6]
Gold-miner(30)	**30**	**30**	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]
Grid(5)	**5**	**5**	**5** [2]	**5** [2]	**5** [2]	**5** [1]	**5** [1]	**5** [1]	**5** [7]
Gripper(20)	**20**	**20**	**20** [1]	**20** [1]	**20** [1]	**20** [1]	**20** [1]	**20** [1]	**20** [2]
Logistics(35)	34	**35**	**35** [6]	**35** [7]	**35** [3]	34 [3]	34 [2]	34 [5]	34 [23]
Matching-bw(30)	20	25	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]
Miconic(150)	**150**	**150**	**150** [1]	**150** [1]	**150** [1]	**150** [1]	**150** [1]	**150** [1]	**150** [2]
Movie(30)	**30**	**30**	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]
Mprime(35)	**35**	**35**	**35** [1]	**35** [1]	**35** [1]	**35** [1]	**35** [1]	**35** [1]	**35** [1]
Mystery(20)	16	**19**	**19** [2]	**19** [2]	**19** [2]	**19** [2]	**19** [2]	**19** [2]	**19** [2]
Nomystery(20)	9	**13**	12 [4]	10 [5]	11 [4]	10 [4]	11 [4]	11 [4]	**13** [9]
N-puzzle(30)	28	**30**	20 [5]	20 [4]	20 [9]	25 [4]	25 [4]	25 [9]	11 [38]
Openstacks(20)	**20**	**20**	**20** [4]	**20** [3]	**20** [2]	**20** [1]	**20** [1]	**20** [1]	**20** [4]
Parcprinter(20)	**20**	**20**	**20** [1]	**20** [1]	**20** [1]	**20** [1]	**20** [1]	**20** [1]	**20** [2]
Parking(20)	**20**	**20**	**20** [4]	**20** [4]	**20** [5]	**20** [2]	**20** [2]	**20** [2]	**20** [7]
Pathways-noneg(30)	**30**	**30**	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [5]
Pegsol(20)	**20**	**20**	**20** [3]	**20** [3]	**20** [2]	**20** [2]	**20** [3]	**20** [2]	**20** [4]
Pipesworld-no-t(50)	43	44	44 [29]	43 [4]	43 [4]	44 [5]	44 [5]	44 [5]	**45** [9]
Pipesworld-t(50)	39	**41**	40 [27]	**41** [7]	**41** [10]	**41** [6]	**41** [6]	**41** [7]	**41** [7]
PSR-small(50)	**50**	**50**	**50** [1]	**50** [1]	**50** [1]	**50** [1]	**50** [1]	**50** [1]	**50** [8]
Rovers(40)	**40**	**40**	**40** [1]	**40** [1]	**40** [1]	**40** [1]	**40** [1]	**40** [1]	**40** [8]
Satellite(36)	**36**	**36**	**36** [4]	**36** [4]	35 [2]	**36** [1]	**36** [1]	**36** [1]	**36** [14]
Scanalyzer(20)	19	**20**	19 [2]	**20** [3]	**20** [3]	**20** [2]	**20** [2]	19 [2]	19 [5]
Sokoban(20)	**19**	17	13 [11]	14 [12]	15 [12]	17 [6]	15 [5]	17 [4]	12 [33]
Sokoban-l(30)	24	21	**30** [5]	**30** [5]	**30** [5]	**30** [4]	**30** [3]	**30** [4]	**30** [15]
Spanner(30)	0	0	0 [0]	0 [0]	0 [0]	0 [0]	0 [0]	0 [0]	0 [0]
Storage(30)	20	19	25 [9]	25 [10]	26 [10]	25 [10]	23 [6]	26 [5]	**27** [22]
Tidybot(20)	13	14	**18** [5]	**18** [4]	**18** [3]	17 [3]	17 [3]	**18** [5]	**18** [6]
Tpp(30)	**30**	**30**	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [1]	**30** [6]
Transport(20)	11	**17**	12 [13]	14 [8]	14 [10]	11 [3]	12 [3]	10 [5]	12 [11]
Trucks-strips(30)	19	18	22 [7]	23 [7]	22 [9]	23 [5]	22 [3]	**24** [4]	22 [10]
Visitall(20)	3	**20**	13 [519]	13 [587]	10 [203]	12 [137]	10 [97]	8 [42]	2 [52]
Woodworking(20)	**20**	**20**	**20** [6]	**20** [5]	**20** [4]	**20** [2]	**20** [2]	**20** [2]	**20** [37]
Zenotravel(20)	**20**	**20**	**20** [1]	**20** [1]	**20** [3]	**20** [1]	**20** [1]	**20** [1]	**20** [1]
Total	1125	1162	1171	1177	1172	1176	1179	1179	1151

Table 6.1: Comparison between FD, LAMA and the different configurations of RPT. Numbers in parentheses represent the total number of problems in the domain. Numbers in brackets represent the number of tree edges that the solution with the highest number of tree edges in that domain has.

are not exploited explicitly by FD during search.

6.6.2 Parameters of RPT

In this work we experiment with the two parameters of RPT ϵ and p to analyze their impact. Higher values of ϵ mean that the local searches are larger, whereas p determines the balance between exploration and exploitation. Prior to the experimentation we expected to have very different results depending on the parameters of RPT; however, Table 6.1 shows that in reality the differences are not that big, even when using a very small ϵ. Although different configurations of RPT have different coverage, their overall behavior compared to FD and LAMA is consistent. Also, there is no overall winner configuration, with different values of ϵ and p being more appropriate in some domains than in others. The configuration with $\epsilon = 1000$ has worse coverage due to the excessively small local searches, but it still obtains the best coverage in a significant number of domains and is not dominated by any configuration.

The exception to the consistent behavior of the different configurations is the *Barman* domain. As mentioned before, sampling in this domain is more complicated due to the frequent generation of spurious states. Because of this, larger values of ϵ and smaller values of p are preferable, since they reduce the sampling tendency of RPT and allow reproducing a behavior closer to FD's and LAMA's.

There are two reasons that explain this behavior. First, most domains from the benchmarks do not contain dead ends, which means that RPT can reach the goals from any node in the tree. If this is the case, as long as some progress is made by expanding towards the goals, RPT will eventually solve the problem despite how irrelevant the sampled states may be. Second, after every *Extend* phase towards a sampled state there is another *Extend* phase towards G, which means that even for very high values of p some effort is still made to advance towards the goals.

6.6.3 Tree Size

Table 6.1 also shows the number of tree edges that the solution with the highest number of tree edges in that domain has. This information is important because it is representative of the size of the tree in the cases in which it is able to scale up to bigger problems. The most obvious conclusion is that RPTs are significantly smaller than the RRTs used in motion planning, as the local searches are computationally much costlier in automated planning. However, in some domains in which the heuristic evaluation is not as costly, like *Visitall*, RPT is able to solve problems by building trees with hundreds of edges. A notable case is the reported 587 edges that RPT_10k_0.5 has to trace back to recover the solution of the hardest instance that it is able to solve in *Visitall*. Actually, several solution plans returned by RPT in *Visitall* have well over 10000 actions, which highlights the robustness and scalability of RPT.

Another relevant fact is that in many cases problems are solved with a single extension even with $\epsilon = 10000$. For example, a total of 946 problems were solved by RPT_10k_0.5 with a single extension out of the 1177 that it can solve overall. This is due to two reasons: first, many of the domains come from old IPCs and thus are relatively easily solved by current state-of-the-art planners; second, most problems in planning are solved either very quickly or not at all due to the exponential blow up of the requirements in time and space as the size of the problems increase, which is also reflected by the size of the tree. As expected, the exception to this is the configuration with $\epsilon = 1000$, which almost consistently has significantly bigger trees than the rest of the configurations.

6.6.4 Sampling

As described in Section 6.5.1, sampling a state in an implicit search space requires solving a CSP. This CSP tends to be very small and the time spent solving it is on average negligible. Some exceptional cases occur though, when sampling a state takes a considerable amount of time. In two problems of *Airport*, two problems of *Tidybot* and in the whole set of *Barman* problems sampling requires more than one second. In particular, for the first four cases it requires 699.55s (problem 46 of *Airport*), 21.88s (problem 47 of *Airport*), 152.53s (problem 19 of *Tidybot*) and 301.75s (problem 20 of *Tidybot*). In the *Barman* domain times range from 0.8s to not being able to sample a state under the time limit of 1800 seconds. However, in all these cases the problem is not the time required to solve the CSP, but rather that G is unreachable from the sampled states - which is detected by the reachability analysis from the sampled state. For example, in problem 46 of *Airport* more than 18000 states had to be sampled before a state from which G could be reached was found, which explains the long time spent sampling.

To ascertain that the heuristics proposed to solve the CSP are indeed necessary, we did some informal experimentation without them. After disabling forward checking and using a random ordering of the *"exactly-1"* invariant groups, in some domains solving the CSP was just not possible. For example, in many instances of *Sokoban* the backtracking algorithm was not able to sample a random state under the time limit of 1800 seconds. Such cases tend to occur in domains in which there are *"exactly-1"* invariant groups highly constrained by mutexes and other *"exactly-1"* invariant groups.

As an additional note regarding the importance of the use of constraints in sampling, we refer the reader to (Alcázar et al., 2011), in which a previous version of this work was presented. In that previous version only mutexes from the monotonicity analysis were used to avoid sampling spurious states. This caused RPT to display a much worse behavior than the planners it was compared with in several domains, hurting significantly the overall coverage of all the analyzed versions of RPT.

6.6.5 Other Measurements

Although memory is usually not a bottleneck in satisficing planning, it is interesting to test whether the claims about the efficiency in terms of memory of RPT are true. In the experimentation we measured the memory necessary to store the tree and the auxiliary structures that allow a faster computation of the heuristic from the nodes of the tree. In all the problems, the amount of memory is smaller than the memory necessary to ground the problem and to compute h^2. This means that the memory needed during search by RPT is in practice determined by ϵ. As a final note and to confirm this fact, during the experimentation FD and LAMA run out of memory before running out of time in 40 and 25 problems respectively, whereas this never happened with RPT.

Regarding quality, the plans returned by RPT were consistently longer than the ones returned by FD and LAMA whenever more than one edge was needed to trace back the solution in RPT. Such is the case in both versions of *Sokoban*, in *Visitall* and in *N-puzzle*. For example, and using the quality score from the last IPC, RPT_10k_0.5's quality score is 8 points lower than FD's in *Sokoban* and very similar in *Sokoban-l* despite it being able to solve 6 problems more in the latter domain. This was to be expected though, as the added exploration and the uniform sampling often cause RPT to take detours on its way to a goal state. The same case was observed with time. Besides, the preprocessing made by RPT can in some cases be longer than the time spent by FD and LAMA during search, which further skews the time score in favor of the latter.

6.7 Related Work

Stochastic search has also been employed by other planners. A prominent example is LPG (Gerevini and Serina, 2002), a planner inspired by random walk search algorithms. LPG searches in the space of plans employing a structure known as the action graph, choosing neighbor graphs based on a parametrized heuristic function. The main relation between LPG and RPT is that LPG tries to balance exploration and exploitation by performing random restarts. These restarts help LPG to avoid exploring plateaus and local minima.

Random exploration has also been proposed in the context of forward-chaining planning. This includes the use of Monte-Carlo Random Walks to select a promising action sequence (Nakhost and Müller, 2009), local search combined with random walks (Xie et al., 2012) and the triggering of random walks when the planner detects that it has fallen into a heuristic plateau (Lu et al., 2011).

Lastly, inspired by real-time versions of RRTs (Bruce and Veloso, 2006), an RRT-like algorithm that stochastically interleaves search and plan reuse has been proposed in (Borrajo and Veloso, 2012). In this case, the increased exploration comes from employing information from past plans instead of from sampled states. This includes reusing both plans and goals from previous searches.

6.8 Conclusions and Future Work

In this work, we have analyzed how to adapt RRTs for their use in automated planning. Previous work has been studied and the challenges that the implementation of RRTs in contexts other than motion planning posed have been presented. In the experimentation we have shown that this approach has much potential, being able to outperform the state of the art. Besides, we have identified the major flaws of this approach, which may allow to obtain better results in the future.

The main challenge of the use of RRTs in automated planning, the design of an algorithm able to sample an implicit search space uniformly, has been thoroughly studied. This novel problem has been tackled by exploiting state invariants of the problem extensively and formulating it as a CSP. The results show that, in domains in which current methods can find most relevant invariants, sampling is both efficient and useful.

Another of the drawbacks of RRTs, the estimation of the closest node, has also been analyzed. Here we employ a more general definition of caching of reachability heuristics, inspired by recent work on regression in automated planning. We have also examined the characteristics of the local planners that can be used along with RPT. We identified their requirements and defined the limit ϵ of the local search in terms of state-space planning.

In the experimental evaluation we have run tests over a broad set of benchmarks. We focused on both overall and *per domain* coverage, with an special emphasis on the defining features of the domains that may affect the performance of RPT in comparison with other state-of-the-art planners. Other measures have been presented, including some specific to RRT-like algorithms like tree size and sampling performance.

After this analysis, several lines of research remain open. First, some approaches like growing two trees at the same time in a bidirectional manner and the implementation of a proper *Connect* phase are still unexplored. Besides, there is abundant research currently done on RRTs related to avoiding pathological cases and introducing biases that allow a more advantageous sampling. These techniques can also be studied for their use in automated planning so the overall performance is improved.

From a planning perspective, techniques like caching the heuristic value of explored states to avoid recomputation when they are expanded several times (Richter et al., 2010) may prove interesting for this kind of algorithms. Another interesting possibility is the usage of portfolios of search algorithms or portfolios of heuristics combined with RRTs in order to compensate the flaws of both best-first search algorithms and RRTs.

As a last remark, another possible future line of research includes adapting this algorithm for a dynamic setting in which interleaving of planning and execution is necessary. This approach looks promising for domains in which exogenous events and partial information may force the planner to replan in numerous occasions.

Part V

Alternative Characterizations of the Landmark Graph

Chapter 7

A SAT Compilation of the Landmark Graph

Landmarks are subgoals formed by sets of propositions that must be achieved at some point in a planning task. These landmarks have some ordering relations between them that form what is known as the landmark graph. Previous works have used information from this graph with several purposes; however, few have addressed some of the shortcomings of the current representation of the graph, like landmarks having to be true at several time steps. In this work we propose a SAT encoding of the landmark graph whose solution represents a more informative version of the original graph.

7.1 Introduction

Landmarks are disjunctive sets of propositions of which at least one component must be achieved or executed at least once in every solution plan to a problem (Hoffmann et al., 2004). Currently, landmarks are a very prominent line of research in automated planning, as the success of landmark-based planners like LAMA (Richter and Westphal, 2010) shows. Most approaches have focused on using landmarks to partition the problem (Hoffmann et al., 2004; Sebastia et al., 2006) or to derive heuristics used in forward search planners (Richter and Westphal, 2010; Karpas and Domshlak, 2009; Helmert and Domshlak, 2009).

Finding the complete set of landmarks or even just proving that a proposition is actually a landmark is PSPACE-complete (Hoffmann et al., 2004). However, current methods can efficiently compute a subset of the landmarks of the task based on a delete-relaxation representation of the problem. A set of partial orders between propositions can be computed with similar techniques too, which applied to landmarks are used to build the landmark graph. The landmark graph is the base of many of the techniques that employ landmarks, as the order in which landmarks should be achieved is often as important as finding the landmarks themselves.

Even though the landmark graph provides important information, it still has several shortcomings. First, the partial orderings may not be enough to come up with a reasonable total order, which is critical for things such as partitioning the problem into smaller subproblems. Second, landmarks appear only once, even when it is clear that they must be achieved several times. For instance, when a landmark $l \in L$ is a causal precondition of other landmarks $l', l'' \in L$ that are mutex between them, l is likely to be reachieved several times. A representative landmark that must be achieved several times for this reason is *(arm-empty)* in the well-known *Blocksworld* domain; *(arm-empty)* appears only once in the landmark graph despite being necessary before every possible *(holding x - block)*, which in turn is mutex with any other possible *(holding y - block)*. Third, the exploitation of the cycles and the unsound reasonable orders in the landmark graph is still unclear. In fact, most techniques that use landmarks are designed to be applied to acyclic graphs, so cycles are usually removed discarding unsound edges first before any kind of search begins.

An important remark is that the concept of time in a total order setting is absent in the landmark graph. In order to introduce time, landmarks must be able to appear as needed at different time steps. For this, they must be labeled with some sort of time stamp. The planning graph (Blum and Furst, 1997) that is often used to represent the planning task does specifically this: every proposition and action is represented several times by nodes labeled with the level they appear at. Hence, landmarks can be represented as several nodes, one per different possible time step, forming a time-stamped planning graph. There is a similar relationship between the orderings in the landmark graph and the different edges of the planning graph too: orderings are constraints that represent causal relationships of precedence between landmarks, and edges in the planning graph are constraints of either causality between actions and propositions or mutual exclusivity.

One of the most popular ways of exploiting the information contained in the planning graph is encoding it into a SAT problem (Kautz and Selman, 1999). Using a SAT solver to find a solution to the encoding allows to find a corresponding solution plan to the problem or to prove that there is none for a given number of parallel time steps. The encoding consists of creating a variable per node in the planning graph and a clause per edge of the planning graph. Given the similarity between the planning graph and the time-stamped landmark graph, this can also be done for the latter. This allows obtaining an enhanced version of the landmark graph that can represent landmarks being required at different time steps and that describes a more consistent total order than the one represented by the original graph. In addition, mutexes will be introduced explicitly as another way of enforcing temporal constraints.

In this work we propose a SAT compilation for the landmark graph. First, previous related work will be analyzed making emphasis on how the information of landmark graph has been exploited. The process of encoding the landmark graph into a SAT problem will be described next, going into detail for every feature relevant to the compilation. Some experimentation will be done to demonstrate the viability of the approach and finally some conclusions will be drawn and a few remarks on future lines of research will be presented.

7.2 Related Work

In this section a series of relevant previous works that have dealt with the landmark graph will be discussed.

7.2.1 Partitioning using Disjunctive Landmarks

Already presented in Section 2.6.1, splitting the original problem into several smaller ones was the original purpose of the landmark graph. Still, we want to make emphasis on how using the landmark graph *as is* may be detrimental to the search process because of its shortcomings. Take for example the classic Sussman anomaly in the *Blocksworld* domain, whose initial state and goal configuration are shown in Figure 7.1.

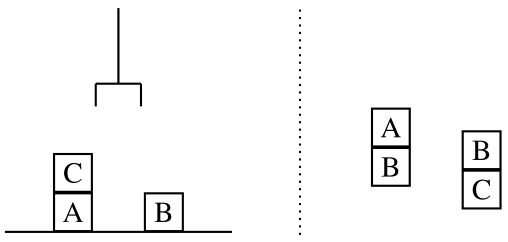


Figure 7.1: Initial state and goal configuration of Sussman's anomaly.

We will follow partially the example shown in Karpaz and Richter's tutorial on landmarks[1] to highlight how a wrong total order can be deduced from the landmark graph. Figure 7.2 shows the initial state and the landmark graph of the problem. Greyed landmarks are landmarks that have been achieved at some point, so landmarks true in I are greyed from the beginning.

Following the method presented in Section 2.6.1, at the first step we have the disjunctive goal $\{$*(clr-A)* $\vee$ *(hld-B)*$\}$. As there is no indication of which proposition should be achieved first, the planner may choose to achieve *(hld-B)* to satisfy the goal by applying *(pick-up B)*. If the process is repeated, the new disjunctive goal would be $\{$*(clr-A)* $\vee$ *(on-B-C)*$\}$. Hence, if *(on-B-C)* is achieved by applying *(stack B C)* we have the state depicted in Figure 7.3.

As we can see in Figure 7.3, there are still several unachieved landmarks. However, in order to achieve them the proposition *(on B C)*, which is a goal of the problem, must be deleted. The following disjunctive goal is $\{$*(clr-A)*$\}$, which is

[1]Landmarks - Definitions, Discovery Methods and Uses. Presented at ICAPS 2010: http://www.icaps10.upf.edu/

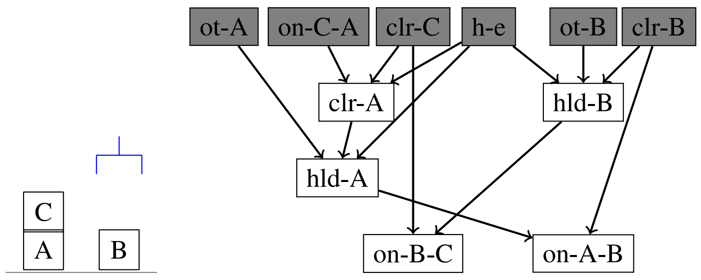


Figure 7.2: Initial state and landmark graph the Sussman anomaly.

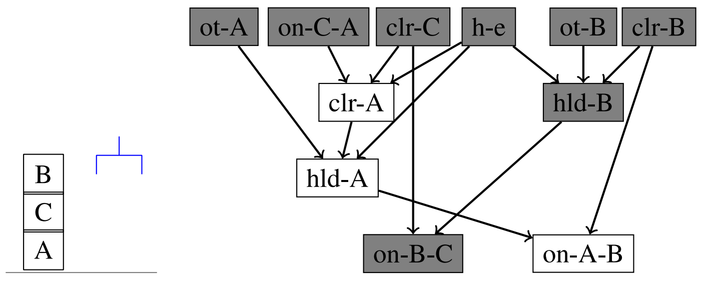


Figure 7.3: Intermediate state and state of the landmark graph in the Sussman anomaly.

achieved by unstacking *B* and *C*. Again, only with this information the planner does not know whether it should reachieve *(on B C)*; this means that a possible plan would be *(unstack B C), (put-down C), (unstack A B), (put-down B)*, which results in the state depicted in Figure 7.4.

The only unachieved landmarks are now *(hld-A)* and *(on A B)*. However, if we achieve them we will find that even if *(on A B)* is true and there are no unachieved landmarks - in the sense that all landmarks have been achieved at least once -, *(on B C)* is not true. In this case *(on A B)* should be deleted and then reachieved after achieving *(on B C)*. All in all in this case following the orderings of the landmark graph and breaking ties randomly may mean wasting a lot of effort, which illustrates how partitioning the problem using the landmark graph may do in some instances more harm than good.

7.2.2 Landmark Layering

A more elaborated way of partitioning the problem is computing layers of conjunctive landmarks, as done in the planning system STELLA (Sebastia et al., 2006). In this case the motivation is the same, but mutexes are taken into account along with the orders to build sets of conjunctive goals. Basically layers are built delaying landmarks that are mutex with other landmarks depending on whether their causal

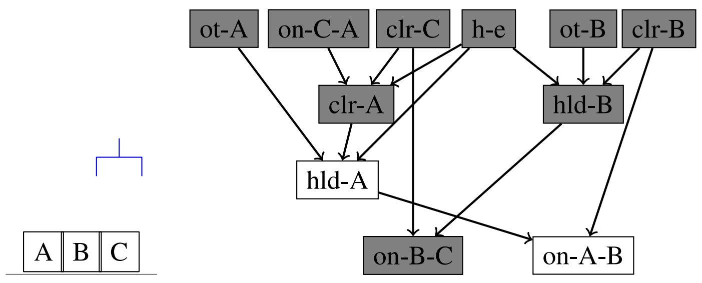


Figure 7.4: Subsequent intermediate state and state of the landmark graph in the Sussman anomaly.

preconditions have been achieved or not. Besides, cycles are dealt with by breaking them and assuming that one landmark must be achieved twice, determining which one and at which time using some heuristic procedures.

This method is more informed, but it is not without disadvantages. First, its computation is substantially more complex, to the point that in some instances the planner may time out even before beginning the search. Besides, it is an empirical approach based on a series of intuitions, which makes its generalization complex.

7.2.3 Temporal Landmarks

With the extensions that appeared in PDDL2.1 (Fox and Long, 2003) time can also be characterized in a planning task. This lead to research on the discovery of temporal landmarks (Sebastia et al., 2007). In this case a temporal planning graph is built after computing the regular STRIPS landmarks and new temporal landmarks are found for a given horizon. The most interesting point is that in the same process both types of landmarks get ”activated” at some point taking into account mutexes and orders. This gives an intuition of when the landmarks (both temporal and STRIPS) may be needed for the first time and even allows proving that there is no solution for a given horizon if some landmark could not be activated in time.

In this work the novel concept of landmarks required at some time step is introduced. However, landmarks still appear as required only once, cycles and unsound orderings are ignored and the whole computation of the activation times depends on a given horizon that is chosen by hand and whose viability is unknown *a priori*.

7.2.4 Landmarks in a CSP to prove Solvability

With the deadline constraints introduced in PDDL3.0 (Gerevini et al., 2009) a hard constraint on the horizon of the planning task can be imposed. A different way of using the landmarks was proposed to detect unsolvable problems due to these time constraints (Marzal et al., 2008). In this case the landmark graph was encoded as a

Constraint Satisfaction Problem (CSP), in which the landmarks were the variables, the time steps where the landmarks might be needed for the first time the domain of the variables and the orders, mutexes and deadlines the constraints. Thus, if the CSP had no solution, the problem was deemed as unsolvable.

In this case a more general approach is taken. Even though the goal is proving unsolvability, probably the most interesting part is the new landmark graph obtained from a solution assignment.

7.3 A SAT Compilation of the Landmark Graph

The planning graph and the landmark graph are two conceptually close concepts. In fact, the planning graph is the core of some landmark discovery methods (Keyder et al., 2010). However, the relationship between the two has not been studied specifically. Landmarks are sets of propositions and actions, so they can be represented in the planning graph as well. The main difference is that landmarks do not contain information regarding time, as opposed to the time-stamped nodes that represent actions and propositions in the planning graph. Hence, the critical point is finding when landmarks are required - and/or how many times - when carrying them over to the planning graph.

Another related concept are the constraints that constitute the edges in both graphs. These edges are causal relationships between the nodes of the graph that encode time constraints. Since time steps are not represented in the landmark graph, its edges are of a more general nature. However, they share most of their properties and can be exploited in similar ways. Following this intuition, it is interesting to analyze whether some of these commonalities allow applying techniques used in the planning graph to the landmark graph.

An interesting approach commonly employed in automated planning is the encoding of the planning graph as a boolean satisfiability (SAT) problem. This is done by planners like Blackbox (Kautz and Selman, 1999) and SATPlan (Kautz et al., 2006), which use a SAT solver to find an assignment that corresponds to a valid solution plan. Despite being a relatively old concept, these planners still represent the state of the art in parallel-length optimal planning and are competitive in satisficing planning, as they effectively take advantage of the techniques developed by the SAT community (Rintanen, 2010). The classical way of translating the planning graph into a SAT instance involves encoding every proposition and action at every level as a variable, and every constraint as a clause. Since the number of parallel steps that the optimal solution has is unknown, the initial number of levels is set to the minimum required for the goal propositions to appear with no mutex relationships between them.[2] Then, the planning graph is converted into a SAT problem and solved using a SAT solver. If no solution is found, the planning graph

[2]Remember that mutexes in the planning graph are dynamic and depend on the level as described in Section 2.5.

is extended by one level and the process is repeated until a solution is finally found or the planning graph levels off, also known as the “ramp-up” method.

Inspired by this approach a SAT compilation of the landmark graph is proposed. In this case, the variables represent the proposition landmarks at every time step, and the clauses their respective ordering constraints. Additionally long distance mutexes (londex) constraints (Chen et al., 2007), which are a generalization of static binary mutexes, are added to the problem. Static binary mutexes indicate that two propositions $p, p' \in S$ cannot be true at the same time; londexes however give a lower bound on the number of parallel steps that must appear between two states $s, s' \subseteq S$ such that $p \in s$ and $p' \in s'$. The reason behind the use of mutexes and londexes is that the landmark graph does not contain explicit information regarding mutual exclusion, which represents a loss of information with respect to the planning graph. The computation of h^{max} from the initial state for every landmark is also required. Landmarks should not be encoded as variables for levels at which they cannot appear, as a minimum number of parallel steps is needed before some propositions can be achieved. This can be found by computing h^{max} and corresponds to the first level at which the landmarks appear in the planning graph.

The same method as with SAT-based planners is used. Once the landmark graph has been built, a SAT compilation for a given number of time steps is computed. Then, a SAT solver is used to find a solution assignment, and, if none is found, the horizon is increased. An important difference is that the “ramp-up” method in combination with the SAT encoding of the landmark graph is not complete, as it cannot prove unsolvability. This is because the SAT compilation of the landmark graph encodes the same constraints independently of the number of levels, whereas the constraints of the planning graph are different per level. Hence, whereas the planning graph can level off and thus prove that a problem is unsolvable, the SAT compilation of the landmark graph cannot.

7.3.1 Clause Encoding

Clauses can be divided in three types: existential clauses, ordering clauses and londex clauses. In these clauses, *ini* represents the time step at which a given proposition $p \in S$ can be true for the first time, found by computing $h^{max}(p)$. *n* is the number of time steps - or horizon - the landmark graph takes into account. Existential clauses represent the fact that the landmarks must be true at some time at least once. They have the following form:

- Existential clauses: Every landmark $a \in L$ must be true at at least one time step $(a_{ini} \vee ... \vee a_n)$

In the particular case of propositions that appear in either I or G these clauses are not necessary, as they always appear at least in the first and last level respectively. The variables that represent these landmarks must be introduced in the problem though, as they may be necessary at different time steps. This way, $\forall p \in I$ then p_{ini} and $\forall p' \in G$ then p'_n.

Ordering clauses represent the edges of the landmark graph. There is a different clause per type of sound ordering:

- Natural orders ($a <_{nat} b$): $a \in L$ must be true at some time step before $b \in L$ is true ($a_{ini} \vee \ldots \vee a_{t-1} \vee \neg b_t$)
- Necessary orders ($a <_n b$): Either $a \in L$ or $b \in L$ must be true at the time step before *b* is true ($a_{t-1} \vee b_{t-1} \vee \neg b_t$)
- Greedy-necessary orders ($a <_{gn} b$): Either $a \in L$ must be true at the time step before $b \in L$ or b must be true at some time step before b is true ($a_{t-1} \vee b_{ini} \vee \ldots \vee b_{t-1} \vee \neg b_t$)

For every landmark order $a < b$ several clauses are needed. In particular, the number of clauses needed depends on the number of levels at which b can be true, from b_{ini} to b_n.

Necessary orderings are worth of mention, as they can induce cycles in the landmark graph. In this case, though, cycles are not undesirable, as they are resolved implicitly when finding a solution assignment. This allows to find important structural information in the planning graph, such as loops or a producer-consumer relationships between propositions. For instance, in the aforementioned Blocksworld domain necessary orders allow to discover that *(arm-empty)* is required at every other time step, as it is necessarily ordered before every *(holding x - block)* proposition.

Reasonable orders behave differently from the rest of the edge constraints. Reasonable orders do not have to be respected in every solution plan in progression, but they must be respected in regression, as proved in Section 3.5.3. In particular, reasonable orders are known to hold among goal propositions (Koehler and Hoffmann, 2000), although this is not limited to them. This means that if $a, b \in S$ and $a <_r b$, the last time *a* is made true must come before the last time *b* is made true. When doing regression, this means that b must be supported first whenever both a and b are true if both must be true until the final state, that is, if they comply with the definition of *aftermatch* (Koehler and Hoffmann, 2000). In terms of defining a constraint, this means that, if *a* and *b* are true at some level t, both remain true until the end of the problem and *a* is reasonably ordered before *b*, then a_{t-1} must be true. The only case in which this is not true is when there exists an action that adds both propositions at the same time, but in that case current methods would not report a reasonable order between them (Richter and Westphal, 2010). Their representation as a SAT clause in the encoding of a landmark graph with *n* time steps is the following:

- Reasonable orderings: If $a, b \in S$, $a <_r b$ and $a_t \wedge b_t$, *a* must be true at the time step before that level if they remain true until *n* ($a_{t-1} \vee \neg a_t \vee \neg b_t \vee \neg a_{t+i} \vee ... \vee \neg a_n \vee \neg b_{t+i} \vee ... \vee \neg b_n$)

Before defining londex clauses, some clarifications must be done. Londexes are a generalization of mutexes that introduce a relation of mutual exclusivity among several time steps. For example, *(holding a)* and *(holding b)* in the Blocksworld domain would form a londex of distance 2, as at least 2 time steps are needed to achieve one of the propositions from a state in which the other is true. Seen as a constraint, this means that both propositions cannot be true neither at the same time step nor at consecutive time steps. A fact that is not mentioned in the definition of londex is that londex are not necessarily symmetrical, in the sense that if $a \in S$ and $b \in S$ are mutex the distance to achieve *b* from a state $s \subseteq S$ in which $a \in s$ may not be the same as the distance to achieve *a* from a state $s' \subseteq S$ in which $b \in s'$. For example, in an instance of the *Sokoban* domain with a single block and in which the grid is a n×n one with no obstacles, a proposition that represents the block being at some corner of the grid and another one that represents the block being somewhere at the center are mutex. However, the minimum number of actions to move the block from the center to the corner is finite, whereas moving the block from the corner to the center is not possible at all (in which case the distance would be considered as infinite). The clauses derived from the londex must take into account this fact, so they are based on a lower bound of the distance between mutex propositions rather than on a single londex constraint:

- Distance between londexes: if $a, b \in S$, *a* cannot be true at a time step *t'* if *b* is true at *t* such that $t - distance(a, b) > t' \leq t$ $(\neg a_{t-(distance-1)} \vee \neg b_t) \wedge ... \wedge (\neg a_t \vee \neg b_t)$

7.3.2 Disjunctive and Conjuctive Sets of Landmarks and Action Landmarks

Landmark discovery methods are not limited to single proposition landmarks. Disjunctive and conjuctive sets of landmark propositions and action landmarks may be found too. Regarding action landmarks, it is easy to see that their preconditions and effects are proposition landmarks themselves. Therefore, actions can be easily encoded with an additional variable with constraints over those landmarks. These constraints can be represented by clauses equivalent to those used when encoding the planning graph as a SAT problem. Two differences exist: first, there are no action levels in the landmark graph, so we must assume that either the preconditions or the effects are true at the same time step than the action; second, preconditions and added propositions are necessarily landmarks, but deleted propositions may not be, so constraints with deleted propositions are generated only if those propositions are landmarks. If that is the case, regular mutex clauses between the action and the deleted propositions would be used. These are the clauses generated if we assume that actions occur at the same time as their effects become true:

- Existential action clauses: Every action landmark $a \in A$ must be true at least at one time step $(a_{ini} \vee ... \vee a_n)$

- Precondition clauses: Precondition $p \in \text{pre}(a)$ must be true one time step before action *a* is true $(p_{t-1} \vee \neg a_t)$
- Add clauses: Added proposition $p \in \text{add}(a)$ must be true at the same time step than action *a* whenever *a* is true $(p_t \vee \neg a_t)$
- *e-delete* clauses: *e-deleted* proposition $p \in \text{del}(a)$ must be false if it is a landmark - $p \in L$ - at the same time step than action *a* whenever *a* is true $(\neg p_t \vee \neg a_t)$

The sets of conjunctive and disjunctive propositions that conform conjunctive and disjunctive landmarks can be represented with auxiliary variables. Existential, natural and greedy-necessary ordering clauses are built for these auxiliary variables - as these are derived from the landmark computation method -, whereas necessary and reasonable orderings and londex clauses are built for the propositions that compose the set. Auxiliary variables are represented in the following way:

- Disjunctive landmarks: If $l_d \in L$ is a disjunctive landmark, at least one proposition $p^i \in l_d$ must be true whenever l_d is true $(p_t^0 \vee ... \vee p_t^n \vee \neg l_{d_t})$
- Conjunctive landmarks: If $l_c \in L$ is a conjunctive landmark, every proposition $p^i \in l_c$ must be true whenever l_c is true $(p_t^0 \vee \neg l_{ct}) \wedge ... \wedge (p_t^n \vee \neg l_{ct})$

7.4 Exploiting the Solution of the SAT Encoding

As described before, the SAT encoding is iteratively generated by increasing its horizon from an initial value. A SAT solver is used for every resulting SAT instance until a solution assignment is found. First of all, it should be noted that the solution may not be unique, there may be several solutions for the same landmark graph encoding; second, if there is a solution for an encoding with a given number of levels *n*, there will be solutions for every encoding with a number of levels greater than *n*. This means that the obtained graph is not sound in the sense that the total order obtained does not have to be respected by every solution plan. Besides, in the case of disjunctive landmarks the propositions that appear as true are not necessarily those that support the disjunctive set for every solution plan.

By solving the problem, a time-stamped landmark graph is obtained. This assignment can be exploited in several ways. The first relevant conclusion is that the number of levels of the graph is a lower bound on the parallel length of the original problem. When regular SAT-based planners that employ the ramp-up method are used this does not represent a great advantage, as their running time is dominated by proving that there is no solution at the level *n-1* when the optimal solution has *n* levels. However, for other approaches that are based on guesses over the minimal parallel length of the problem (Xing et al., 2006; Rintanen, 2010) this can be used to reduce the range of horizons considered.

Another advantage of the time-stamped landmark graph is that a plausible guess over the time steps at which a landmark may be necessary is obtained. This allows the construction of a roadmap that can be used to guide the search. Closely related with this idea is the aforementioned partitioning done by STELLA of the problem by using layers of conjunctive landmarks described in Section 7.2.2, although in this case the layers can be built just by choosing the landmarks that appear as true at every particular level with no additional computation.

An important difference with the original graph is that here landmarks may appear as true several times. This can mean two things: first, a landmark must not only be achieved but also may be kept as true across different levels; second, it is often the case that a landmark $l \in L$ appears as true at non-consecutive time steps t_i, t_j and there is another landmark $l' \in L$ mutex with *l* which appears as true at some level t' such that $t_i < t' < t_j$. If this can be proven for every solution plan this means that *l* must be achieved, deleted and then achieved again. In conjunction with landmark-based heuristics this would allow to obtain more informative values; in particular, admissible landmark-based heuristics would keep their admissibility while not being limited by the upper bound that h^+ imposes to admissible delete-effect relaxation heuristics.

7.4.1 An Example of Time-Stamped Landmark Graph

In Figure 7.2 the landmark graph of the Sussman's anomaly is shown. Although the graph contains the most relevant propositions and orderings, it gives little insight on what the structure of the task may be like. In particular, the cycles induced by the necessary orders on *(arm-empty)* - which represent the concept of the arm as a common resource in *Blocksworld* - are absent. This is so because common techniques that exploit the landmark graph require it to be acyclic.

Table 7.1 represents the output of the encoding process. LAMA (Richter and Westphal, 2010) was used to generate the landmark graph, in particular with using Zhu and Givan's method (2003). Changes were done to its algorithm to allow the computation of necessary orders, as LAMA does not compute them. Besides, cycles in the landmark graph were not removed. The table shows at which levels landmarks must be true to satisfy the constraints. The opposite is not true; a landmark does not have to be false when it appears as not required. That a landmark must be false may be useful in some cases, although this is trivially computable using the original constraints along with the solution.

This solution is a good example of how some of the shortcomings of the landmark graph can be overcome by employing the proposed method. First, the number of levels is the minimum required; second, landmarks like *(arm-empty)* required at several levels are detected; third, the positions at which landmarks appear as needed offer a great deal of information with regards to possible solutions. In this notable case, the optimal solution plan always respects the required landmarks at the exact times; seen the other way around, the only sequence of actions that would comply with the solution of the compilation of the landmark graph is the optimal plan.

Level:	0	1	2	3	4	5	6
(arm-empty)	x	-	x	-	x	-	-
(on a b)	-	-	-	-	-	-	x
(on b c)	-	-	-	-	x	x	x
(on c a)	x	-	-	-	-	-	-
(clear a)	-	x	x	x	x	-	-
(clear b)	x	-	x	-	-	x	-
(clear c)	x	-	x	x	-	-	-
(on-table a)	x	-	-	-	-	-	-
(on-table b)	x	-	-	-	-	-	-
(holding a)	-	-	-	-	-	x	-
(holding b)	-	-	-	x	-	-	-
(holding c)	-	x	-	-	-	-	-

Table 7.1: Output of the solution of the SAT problem generated from the landmark graph.

7.5 Experimentation

Although this work is centered around an alternative method of exploiting the information contained in the landmark graph as a departure point for novel approaches, an experimental part has been added for the sake of completeness. In this experiments the landmark layering approach used by STELLA (described in Section 7.2.2) has been emulated in the following way:

- The landmark graph is compiled into a SAT problem.
- A solution for the resulting SAT encoding is found using the "ramp-up" method employed by conventional SAT solvers.
- The problem is partitioned by creating a series of ordered subgoals. Subgoals are sets of conjunctive landmarks that appear as needed in a given layer of the new time-stamped landmark graph. These subgoals are ordered by the level they were extracted from.
- A base planner is used to solve the different subproblems. The initial state I for every subproblem is the final state $s \supseteq G$ from the previous one. The final solution is the concatenation of the solution plans of all the subproblems.

The base planner is Fast-Downward (Helmert, 2006) with greedy best-first search as the search algorithm, the FF heuristic and preferred operators with boosting enabled. Experiments have been done on a Dual Core Intel Pentium D at 3.40 Ghz running under Linux. The maximum available memory for the planner was set to 1GB, and the time limit was 1800 seconds. Only non-disjunctive propositional landmarks were used in the encoding of the landmark graph. No parameter setting was necessary.

Experiments were done using the same domains that were used when STELLA was compared to other planners: Blocksworld, Miconic, Freecell, Logistics, Depots, Driverlog, Satellite and Zenotravel from the second and the third international planning competitions. Mutexes were computed using the invariant discovery process of Fast-Downward. Since this method is unable to detect important mutexes in some domains, two domains were left out: Depots and Freecell. Elevators was also left out because single propositional landmarks in this domain do not offer additional information.

Regarding coverage (number of problems solved), the base planner and the partitioning approach were able to solve the same instances. This is due to the fact that the base planner is able to solve all the instances - using a less efficient base planner, as done by STELLA, could be interesting to see if there is a significant improvement -. Therefore, we will focus on quality and number of expanded nodes to compare both approaches.

Table 7.2 shows the comparison between the base (Base) planner and the partitioning approach (Part) in terms of evaluated states (-S) and solution quality (-Q) in the Blocksworld domain. Instances that were solved by both approaches expanding fewer than 100 nodes were left out, as they are deemed too easy to contribute with relevant information. Results show that in many cases partitioning leads to an increase in quality. Instances in which the partitioning approach finds shorter plans usually require fewer nodes expansions for this approach as well. The opposite phenomenon can be observed too: when the base planner finds shorter plans the number of states is also smaller. In this case, the differences are greater, which explains why the average number of expanded states by the partitioning problem is considerably bigger.

In the other domains the results are not so conclusive. The geometric mean of the number of evaluated nodes and the plan length was computed for every domain and the ratio (the mean of the base planner divided by the mean of partitioning approach) displayed. Table 7.3 shows these results. On average the number of expanded nodes is greater, whereas quality remains the same. This is explained by two factors: first, the base planner is already very competitive and so there is little margin for improvement; and second the usage of preferred operators with the FF heuristic already guides the search towards unachieved landmarks, which overlaps with the usage of the landmark graph. A partitioning scheme with a planner based on other paradigms, like LPG (Gerevini and Serina, 2002) and VHPOP (Younes and Simmons, 2003), would probably yield more positive results.

7.6 Conclusions and Future Work

In this work an alternative representation of the landmark graph has been proposed. We have discussed the shortcomings of the original landmark graph and analyzed

Problem:	Base-S	Part-S	Base-Q	Part-Q
Prob-9-0	145	112	72	28
Prob-9-1	141	108	60	28
Prob-9-2	73	104	44	26
Prob-10-0	228	136	60	34
Prob-10-1	148	128	60	32
Prob-10-2	145	136	52	34
Prob-11-0	98	2994	54	72
Prob-11-1	327	628	104	64
Prob-11-2	150	3038	72	62
Prob-12-0	269	30836	74	125
Prob-12-1	131	248	102	52
Prob-13-0	678	284	108	68
Prob-13-1	391	674744	108	120
Prob-14-0	238	5968	88	140
Prob-14-1	268	390732	94	342
Prob-15-0	1504	11052	184	144
Prob-15-1	498	604	124	114
Prob-16-1	455	514	102	96
Prob-16-2	2006	29034	160	118
Prob-17-0	2424	190	280	48
Geometric Mean	299.91	1419.31	87.32	68.30

Table 7.2: Comparison between the base planner and the partitioning approach in terms of evaluated states (columns ”Base-S” and ”Part-S”) and solution quality as plan length (columns ”Base-Q” and ”Part-Q”).

previous related work that has exploited the information it represents. The points in common between the planning graph and the landmark graph have been brought up and a parallelism between the SAT encoding of the constraints of both graphs has been presented. Finally, we have described the SAT encoding of the landmark graph and analyzed its characteristics.

We have also proposed several ways of exploiting the new landmark graph that can lead to future lines of research. Some of the information obtained after solving the SAT problem generated by the landmark graph, like the minimum number of levels, can be straightforwardly used with current techniques. The structure of the graph itself can also be exploited, as it has been done in the experimentation section or in alternative ways, for example by using it as the seed for local search planners like LPG. Also the fact that landmarks can appear several times may lead to more accurate landmark-based heuristics for forward search planners both in satisfying and optimal search. Finally, generalizing the SAT encoding of the landmark graph

Problem:	Mean States	Mean Quality
Driverlog	0.89	1.06
Logistics	0.8	0.87
Satellite	0.45	1.01
Zenotravel	0.54	1

Table 7.3: Comparison between the base planner and the partitioning approach for the rest of the evaluated IPC-2 and IPC-3 domains

for temporal and numeric domains by transforming it into a constraint satisfaction problem (CSP) is also a promising continuation of this work.

Chapter 8

Generalization of the Landmark Graph as a Planning Problem

Landmarks are logical formulæ over sets of propositions or actions that must be satisfied at some point in a planning task. The landmark graph, a proposed representation of the set of landmarks and their interactions, is built using the landmarks of the task and ordering relations between them. A formalization of the landmark graph in terms of a planning task has not been proposed yet, which makes it difficult to determine the significance of the landmark graph with respect to the original planning problem. In this work we propose a generalization of the landmark graph as an abstraction of the original problem and analyze its characteristics.

8.1 Introduction

As analyzed in Chapter 7, landmarks are one of the most important lines of research in automated planning. Nevertheless, the landmark graph still has some limitations in its current form. First, an informative total order may not be straightforwardly deduced from the set of partial orders, which is critical in applications like factored planning; second, landmarks may have to be achieved several times in every solution plan, which is not taken into account due to the lack of negative interactions between landmarks other than the causal orders (Hoffmann et al., 2004); third, the exploitation of the cycles in the graph is still unclear and so current techniques usually just remove them. Furthermore, although the landmark graph has been exploited empirically, a more thorough theoretical analysis that may relate it to the original planning task has not been performed yet.

To address these shortcomings, in this work a generalization of the landmark graph as a planning problem is proposed. The main motivation is creating an automatic landmark-based abstraction whose solution can capture the causal structure of the problem in a more accurate way than the landmark graph. The idea presented in this work is using the landmarks to build the set of propositions of the problem and transforming the achievers of those landmarks into the actions of the abstraction, adding information from landmark orders and mutexes. This abstraction is a planning task itself, so it can be solved just like any other planning problem. Besides, it has several properties of its own that can be helpful when solving the original task.

8.2 Generalization of the Landmark Graph as a Planning Problem

In this section, the process of generating a new problem from the original problem using the information of the landmark graph is described.

8.2.1 Definition of the New Problem P_{abs}

Here, we consider only single proposition landmarks; other cases will be analyzed in a Section 8.2.5. Given $P = (S, A, I, G)$, the generalization of the landmark graph as a planning problem is a tuple $P_{abs} = (S_{abs}, A_{abs}, I_{abs}, G_{abs})$ where S_{abs} is a set of propositions derived from the discovered proposition landmarks, A_{abs} is a set of grounded actions derived from the achievers of the propositions contained in S_{abs}, $I_{abs} \subseteq S_{abs}$ is the value of S_{abs} in the initial state and $G_{abs} \subseteq S_{abs}$ the set of goal propositions.

The set of propositions S_{abs} contains two propositions per landmark $l \in L$, one proposition l_{abs}indicating whether the landmark is true and another proposition $achieved(l_{abs})$ indicating whether the landmark l_{abs} has been previously achieved.[1] All the propositions of the original problem that are not landmarks are discarded, that is, $S \setminus L$ are not taken into account in P_{abs}. The set of actions A_{abs} are the actions in A that are landmark achievers, that is, at least one landmark proposition appears in its *add* effects. Formally, if $a \in A$ and $add(a) \cap L \neq \emptyset$, then a is used to derive new actions in A_{abs}. A priori, a single new action per achiever in A is created. However, due to the possibility of having disjunctive preconditions in the new actions, some actions in A_{abs} may be split into several ones. The details of how to create the new actions of A_{abs} are presented in subsequent sections. All the actions in A that do not add at least one landmark proposition are ignored. The goal propositions G_{abs} are the goal propositions G of the original problem, since a goal proposition is a landmark by definition. Additionally, the propositions that encode whether the original goal propositions have been achieved can also be

[1]A more concise representation using multi-valued state variables could be obtained in two ways: first, the variables could take the values of *true*, *false* or *false but previously achieved* instead of using two different variables to avoid redundancy, although this can lead to actions with disjunctive preconditions; second, landmarks belonging to the same variable in the original problem or that are otherwise mutually exclusive could be grouped in a single variable to indicate whether they are true or not.

added to G_{abs}, since a proposition being true necessarily implies that it has been achieved.

P_{abs} is a regular planning task, so it has all the properties of a regular planning problem. Besides, P_{abs} is an abstraction of the original problem P. The function α that maps a state $s \subseteq S$ to $\alpha(s) \subseteq S_{abs}$ consists of determining which landmarks are true or have been previously achieved in s. Note that this depends not only on s but also on the path that led from I to s, so the information about which landmarks have already been achieved must be stored in a similar fashion as when computing h_{LA}. The transitions are defined by the relationship between A_{abs} and A, described in the following subsections.

8.2.2 Preconditions of the New Actions

Actions in A_{abs} are applicable when the landmarks appearing in their preconditions have the required value. Three different cases define the preconditions of the actions:

- Preconditions derived from the orders between the landmarks.
- Preconditions of the achievers in the original problem that are landmarks.
- Preconditions obtained from actions labeled as *late achievers*.

The orderings of the landmark graph encode the causal relationships between the landmarks. To represent this in P_{abs}, new preconditions are added to the actions in A_{abs}. These preconditions enforce the partial orders between landmarks without hindering the computation of a valid total order. Each type of order implies a different set of new preconditions:

- Natural order: Every achiever $a \in A$ of a given landmark *l* has a precondition for every natural order which represents that the landmarks naturally ordered before *l* must have been previously achieved. Formally, if $l \in add(a)$, $\forall l' <_{nat} l : achieved(l') \in pre(a)$.
- Necessary order: Every achiever $a \in A$ of a given landmark *l* has a precondition for every necessary order which represents that the landmarks necessarily ordered before *l* must be true. Formally, if $l \in add(a)$, $\forall l' <_{n} l : l' \in pre(a)$.
- Greedy-necessary order: Every achiever $a \in A$ of a given landmark *l* has a precondition for every greedy-necessary order which represents that the landmarks greedy necessarily ordered before *l* must be true or that *l* has been previously achieved. Formally, if $l \in add(a)$, $\forall l' <_{nat} l : (achieved(l) \vee l') \in pre(a)$.

Greedy-necessary orders add a disjunctive precondition. Most planners do not support disjunctive preconditions, so actions with disjunctive preconditions must be split into several actions. In theory this can lead to having 2^n new actions per action, where n is the number of disjunctive preconditions. However, due to the dominance that often appears between actions (described in a subsequent subsection), the actual number of new actions is at most $2^{n'}$ actions, where n' is the number of landmarks with greedy-necessary orders achieved by the action. Although still exponential, in practice it seldom happens that an action achieves more than 2 such landmarks, which explains why there is no exponential blow-ups of the size of P_{abs} in the current benchmarks.

Preconditions can also be derived from the landmark preconditions of the actions in A: occurrences of landmarks in the actions in A must be considered when generating the actions in A_{abs}. This means that the preconditions of the original actions in A must also appear in the actions in A_{abs} if they are landmarks: if $a \in A$ was used to create $a' \in A_{abs}$, then $pre(a) \cap L \subseteq pre(a')$. This overlaps with the preconditions obtained from greedy-necessary orders and strictly dominates those obtained from necessary orders, hence making the computation of the latter not necessary. This is so is because both types of orders are computed from common preconditions of the achievers of the landmark, already taken into account this way.

Additionally, there are two kinds of achievers (as described in Section 2.6): *first achievers*, which can achieve some landmark when it has never been achieved before, and *late achievers*, which can achieve a landmark only if it has been already achieved at some time before. Hence, a late achiever $a' \in A_{abs}$ derived from action $a \in A$ has an additional precondition $achieved(l) \in pre(a')$ per landmark $l \in add(a) \cap L$ if a is a late achiever of l.

8.2.3 Effects of the New Actions

Effects of the original achievers and *e-deletion* determine the effects of the new actions. First, the *add* effects of the actions in A that are landmarks can be straightforwardly encoded in the new actions belonging to A_{abs}. Thus, every landmark proposition added by an action in A appears also as an *add* effect in the actions created from it in A_{abs}, whereas non-landmark propositions added by the action are ignored. Similarly, for every regular *add* effect of a landmark l, an *add* of $achieved(l)$ must be included. In summary, if $a \in A$ was used to derive $a' \in A_{abs}$ and landmark $l \in add(a) \cap L$ then $l \wedge achieved(l) \in add(a')$.

Second, negative effects are linked to the notion of *e-delete* presented in Section 2.5.3. Every landmark that is *e-deleted* by an action $a \in A$ is added to the new action in A_{abs} as a *delete* effect. There is an exception to this rule, though: if an action $a \in A$ *e-deletes p* because it has a precondition mutex with *p* and does not add *p* (second condition of *e-deletion*) and that precondition is a landmark, it means that the new action in A_{abs} is only applicable *iff p is false* as long as *e-deleted* fluents are encoded as negative effects in all the actions of A_{abs}. In this case *p* will be always false after the execution of the new action in A_{abs}, which means there is no need to explicitly delete it.

8.2.4 Dominance Between the New Actions

Due to the abstraction of non-landmark propositions and the splitting of disjunctive preconditions, it is possible to have actions in A_{abs} that are equivalent to or dominated by other actions. In particular, an action with the same effects and with a cost greater than or equal to another action needs not to be included in A_{abs} as long as its preconditions are a superset of the preconditions of the dominating action. Formally, $a \in A_{abs}$ dominates $a' \in A_{abs}$ if $a \neq a' \land add(a) = add(a') \land del(a) = del(a') \land pre(a) \subseteq pre(a') \land cost(a) \leq cost(a')$. The pruning of dominated actions can be done after generating all the actions to avoid redundancy and obtain a smaller instance of P_{abs}.

To illustrate this, a small example will be given. In most transportation domains, such as *Logistics*, there is an action that involves an agent moving to a location. If an agent a_i being at some location loc_j, that is, $(at\ a_i\ loc_j)$, is a landmark and a_i being at every adjacent location to loc_j is not a landmark, then all the different *move* actions will have the same preconditions and effects. In this case, keeping all the actions does not add information to P_{abs} and only the one with minimum cost (or a random one among those with minimum cost) is kept.

Dominance between actions also explains why splitting actions when using greedy-necessary orders leads to having fewer additional actions than expected. For example, let's assume that an action $a \in A$ adds a landmark $l \in S$ which is greedy-necessarily ordered after n landmark propositions $p_i \in S$. This means that the new action $a' \in A_{abs}$ will have n disjunctive preconditions of the form $\{achieved(l) \lor p_i\}$. If a' is split into two new actions for every disjunctive precondition, a total of 2^n new actions will be created. However, every time an action is split, either $achieved(l)$ or p_i will be added to the preconditions; if $achieved(l)$ is added and it is already present, no new action is needed, and if p_i is added and $achieved(l)$ is already a precondition, it will be forcibly dominated by another action with $achieved(l)$ as precondition. In the end, only 2 actions are generated per added landmark with greedy-necessary orders, one with $achieved(l)$ as additional precondition and another one with $p_1 \land ... \land p_n$ as additional preconditions.

8.2.5 Conjunctive Landmarks, Disjunctive Landmarks and Action Landmarks

Landmarks are not restricted to the single proposition case. When generalizing the landmark graph this has to be taken into account. Particular cases are treated in the following way:

- Action landmarks: They can be safely ignored, as their preconditions and effects are landmarks themselves. A possible optimization is allowing only the action landmark as achiever if the landmarks derived from the effects of the action are not ordered after any landmark other than the preconditions of the action landmark.

- Conjunctive landmarks: All the propositions that form the set can be treated as independent landmarks. The achievers of landmarks ordered after a conjunctive landmark will have all the propositions in the set as preconditions. Formally, if $l_c, l' \in L$, l_c is a conjunctive landmark and $l_c < l'$, then $\forall a \in A_{abs}$ such that $l' \in \text{eff}(a)$ then l_c must be encoded in $\text{pre}(a)$ accordingly depending on the type of landmark order $l_c < l'$.

- Disjunctive landmarks: All the propositions that form the set can be treated as independent landmarks. Only one of the propositions that form the set must have the required value for an action $a \in A_{abs}$ that has the disjunctive landmark as precondition to be applicable. This is done by splitting every such action into several actions, one per proposition in the set, and including that proposition as precondition. This can lead to an exponential number of actions, so a more efficient way of dealing with disjunctive landmarks could improve the compactness of the problem. Formally, if $l_d, l' \in L$, l_d is a disjunctive landmark and $l_d < l'$, $\forall a \in A_{abs}$ such that $l' \in \text{eff}(a)$ a must be split in $|l_d|$ actions and $\forall l_i \in l_d$ then l_i must be encoded in $\text{pre}(a_i)$ accordingly depending on the type of landmark order $l_d < l'$.

8.3 Properties of the New Problem

Due to its relationship with the original problem P, P_{abs} has several additional characteristics. In this section we present them as a series of theorems.

Theorem 5. *All the propositions in S_{abs} are landmarks in P_{abs}, unless they were generated from a disjunctive landmark of L. If this is the case, they will be part of a disjunctive landmark if a landmark discovery method that can find disjunctive landmarks of size equal or greater than the original disjunctive landmarks in L is used in P_{abs}.*

Proof. All the landmarks in L must be achieved at some point in S. If an artificial proposition per landmark $l \in L$ of the form *achieved(l)* is introduced such that $\forall a \in A$ and $l \in \text{add}(a)$ then $\text{achieved}(l) \in \text{add}(a)$. As all the landmarks in L must be added, all the propositions of the form *achieved(l)* will be true in every goal state and thus can be added to G. Since $G_{abs} = G$, then the propositions of the form *achieved(l)* can be part of G_{abs}. As the propositions of the form *achieved(l)* are only made true whenever its correspondent $l \in S_{abs}$ is made true, then $\forall l \in S_{abs}$ l is a landmark. □

Theorem 6. *The optimal cost to the goal in the new problem h^*_{abs} is an admissible estimation of the cost to the goal in the original problem h^*, since P_{abs} is an abstraction of P.*

Proof. P_{abs} is a projection of P except for the added propositions of the type $achieved(l)$ for $l \in L$. If a problem β' is a projection of β, then $h^*_{\beta'} \leq h^*_{\beta}$. Hence, the only source of inadmissibility can be the propositions of the type $achieved(l)$. However, these propositions are added to enforce the orders of the landmark graph. These orders are respected by any optimal solution of β, so the presence of these propositions cannot make that $h^*_{\beta'} > h^*_\beta$. □

Theorem 7. *h^*_{abs}, when being used as a heuristic in a forward search algorithm, is* not *necessarily a consistent estimation of h^*, as I_{abs} depends on the path that led to the current state in P. That is, h^*_{abs} as an admissible estimation of the cost to the goal is a path-dependent heuristic.*

For the proof, we refer the reader to (Karpas and Domshlak, 2009).

Theorem 8. *h^*_{abs} dominates the admissible landmark counting heuristic h_{LA} (Karpas and Domshlak, 2009) and is not bounded by h^+, the optimal cost of the delete-relaxation version of the original problem.*

Proof. On one hand, if $S = L \subset S_{abs}$, then $h^*_{abs} = h^*$ and $h^*_{abs} \geq h^+ \geq h_{LA}$. In that case, if some landmark $l \in L$ must be deleted and then reachieved with an action $a \in A \mid const(a) > 0$, then $h^*_{abs} > h^+$; on the other hand, if P_{abs} is relaxed by removing orders and *deletes*, then $h^*_{abs} \geq h^{opt}_{LA}$, as P_{abs} then becomes a hitting set problem with respect to L, problem of which h^{opt}_{LA} is the continuous relaxation. There is no further relaxation other than projecting landmarks away that may make h^*_{abs} lower, so h_{LA} is a lower bound of h^*_{abs} if all the landmarks are considered when building P_{abs}. □

Theorem 9. *The optimal cost of the delete-relaxation version of the new problem h^+_{abs} still dominates h_{LA}, even with optimal cost partitioning (h^{opt}_{LA}).*

Proof. Landmark preconditions of the achievers add additional constraints that may make the cost of the optimal plan go beyond h_{LA}'s value. An example is shown in the delete-free problem appearing in Figure 8.1: if $I = \{l\}$ and $G = \{l', l''\}$, h_{LA} would assume a cost of 1 for both l' and l'' for a total cost of 2; h^+_{abs} however takes into account that to achieve l' with a cost of 1 l'' must be achieved first, so h^+_{abs} would yield a value of 3. □

Theorem 10. *The optimal cost of the delete-relaxation version of the new problem h^+_{abs} is dominated by the optimal cost of the delete-relaxation version of the original problem h^+.*

Proof. The proof is analogous to that of Theorem 6. □

8.4 Example

In this section we show an example of how P_{abs} is constructed. The example is taken from a variation of the *Logistics* domain, in which case trucks can only carry

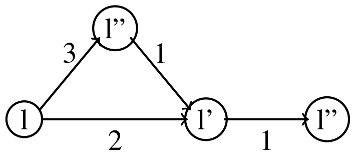


Figure 8.1: Example in which h^+_{abs} yields a higher value than h_{LA}.

one package at a time, encoded by the predicate *(empty ?t - truck)*. The initial state is shown in Figure 8.2. There is a single truck that can move through a graph of different locations. The truck is initially at location *A*, where there is an arbitrary number of packages. The packages can be loaded into the truck if the truck is empty and dropped at another location. The goal is carrying all the packages to the goal location *G*. The single proposition landmarks of this problem are:

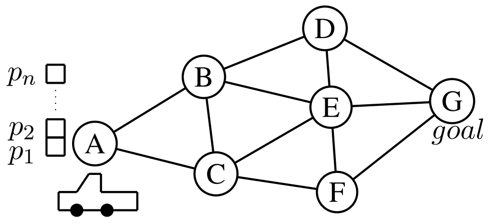


Figure 8.2: Initial state of the modified *Logistics* instance.

- The truck at the initial location: *(at truck A)*
- The truck at the final location: *(at truck G)*
- The truck empty: *(empty truck)*
- Every package p_i at the initial location: *(at p_i A)*
- Every package p_i in the truck: *(in p_i truck)*
- Every package p_i at the final location: *(at p_i G)*

The actions would be the following (note that every time a landmark is made true, the proposition that encodes whether it has been achieved before is unconditionally made true as well):

- Move the truck to the initial location *A*: *pre*=\{*achieved(at truck A)*\}, *add*=\{*(at truck A)*\}, *del*=\{*(at truck G)*\}
- Move the truck to the final location *G*: *pre*=\{*achieved(at truck A)*\}, *add*=\{*(at truck G)*\}, *del*=\{*(at truck A)*\}

- Load a package p_i at the initial location A: *pre=*$\{$*(at truck A),(at* p_i *A),(empty truck)*$\}$*, add=*$\{$*(in* p_i *truck)*$\}$*, del=*$\{$*(at* p_i *A),(empty truck)*$\}$

- Load a package p_i at the final location G: *pre=*$\{$*(at truck G),(at* p_i *G),(empty truck)*$\}$*, add=*$\{$*(in* p_i *truck)*$\}$*, del=*$\{$*(at* p_i *G),(empty truck)*$\}$

- Drop a package p_i at the initial location A: *pre=*$\{$*(at truck A),(in* p_i *truck)*$\}$*, add=*$\{$*(at* p_i *A),(empty truck)*$\}$*, del=*$\{$*(in* p_i *truck)*$\}$

- Drop a package p_i at the final location G: *pre=*$\{$*(at truck G),(in* p_i *truck)*$\}$*, add=*$\{$*(at* p_i *G),(empty truck)*$\}$*, del=*$\{$*(in* p_i *truck)*$\}$

In this example all the preconditions are taken from the landmark preconditions of the original actions in A with the exception of the *move* actions. Similarly, apart from the *move* actions all the actions in A_{abs} can only be generated from a single action in A. The *move* action that achieves *(at truck A)* can be generated from two actions in A; however, both actions have the same preconditions and effects, so one arbitrarily dominates the other, which explains why there is only one action in A_{abs} that achieves *(at truck A)*. This action has *achieved(at truck A)* as precondition because the original actions in A are *late achievers* of *(at truck A)*. No other precondition appears, as no precondition of the original actions is a landmark and *(at truck A)* is not ordered after any other landmark. The same dominance occurs with the action that achieves *(at truck G)*, although in this case the precondition is generated from the natural orders existing between *(at truck A)* and *(at truck G)*. As a side note, every landmark of the form *achieved(s)* that is true in I_{abs} is a static fact, so it can be safely discarded, as with *achieved(at truck A)* in the example.

To better understand the resulting abstraction P_{abs}, Figure 8.3 shows a scheme of what P_{abs} would look like. As we can see all non-landmarks locations have been abstracted away and the actions that move the truck in A_{abs} behave as shortcuts through the abstracted location graph.

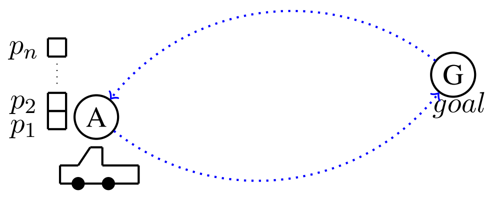


Figure 8.3: Graphical scheme of the resulting abstraction P_{abs}.

The key feature that regular delete-relaxation approaches do not capture in this case is achieving and deleting *(at truck A)* and *(at truck G)* alternatively. In this case, if n encodes the number of packages, the cost of the optimal solution in the original problem is $h^* = 8(n-1)+5$ and in the abstraction is $h^*_{abs} = 4(n-1)+3$. On the other hand, the value of both h^+ and h^{lmcut} (Helmert and Domshlak, 2009) in the original problem is $h^+ = h^{lmcut} = 2n+3$ and the value of the admissible landmark heuristic with optimal cost partitioning h^{opt}_{LA} is in both the original and the delete-relaxation problem $h^{opt}_{LA} = 2n+1$.

8.5 Potential Applications of P_{abs}

Although beyond the scope of this work, it is interesting to analyze the possible applications of P_{abs} in regards to task solving. P_{abs} is a planning problem itself and so it is PSPACE-complete in the general case (Bylander, 1994), which means that in many cases solving it (optimally or otherwise) may not be tractable. However, P_{abs} is in most cases smaller than P, so if a planner is not able to solve P_{abs} it is highly unlikely that it would be able to solve P.

One of the first applications of landmarks, factored planning (Hoffmann et al., 2004), seems to be a technique that would greatly benefit from this approach. As opposed to using the first layer of unachieved or required again landmarks as a disjunctive intermediate goal as described in Section 2.6.1, suboptimally solving P_{abs} can be used to obtain a total order of single landmarks, which can be used as subgoals to partition P. Apart from being able to exploit cycles between landmarks and offering a finer partitioning, this approach has also the advantage of not being subject to arbitrary decisions when dealing with disjunctive subgoals - for instance, when partitioning the Sussman's anomaly with a disjunctive goals as depicted in Figure 7.2, Figure 7.3 and Figure 7.4 -.

Inspired by the potential of the serialization of the landmark graph, the recent planner PROBE (Lipovetzky and Geffner, 2011) builds probes that try to achieve the goal with little to no search by estimating a total order of the landmarks and trying to achieve them following that order in a greedy way. However, the way the probes are built is not based on any theoretical scheme and it has the additional problem of not allowing actions that delete a landmark to be used if that landmark is a goal or is still needed - which is often the case when cycles between landmarks appear -. Solving P_{abs} suboptimally yields a total order that may be more informative and that may capture stronger interactions between landmarks, producing a more informative probe.

The use of P_{abs} to derive heuristics for P has already been mentioned. Nevertheless, solving P_{abs} at every state to obtain the heuristic estimate would be impractical in most cases, so more synergistic techniques are required. Since P_{abs} is an abstraction of P, hierarchical search algorithms like Hierarchical A* (Holte et al., 1996) can be an interesting alternative. When using HA*, P_{abs} is solved from I_{abs} to obtain a heuristic estimate and the expanded nodes are kept hoping that subsequent queries can be cached, which allows obtaining $h(s)$ with no search in P_{abs}. If cache hits happen often enough, there will be a trade-off between the time spent solving P_{abs} and the time saved from computing $h(s)$, which may result in an increased efficiency. Furthermore, P_{abs} can be solved both optimally, superseding the admissible h_{LA}, or suboptimally, superseding LAMA's heuristic (Richter and Westphal, 2010).

Lastly, the total order obtained from solving P_{abs} can also be used in combination with other search paradigms. For instance, local search planners like LPG (Gerevini and Serina, 2002) are often ill-suited to solving highly sequential domains, particularly if they involve cycles. In this case, the solution to P_{abs} can be used as a seed so a skeleton of the plan is provided and the local search only has to fill the "gaps" between the landmarks.

8.6 Experimentation

As any other landmark-based technique, P_{abs} is highly dependent on the number and relevance of the landmarks found with current methods. For instance, in some domains only a few landmarks other than the propositions true in I and the goals in G are found, in which case landmark-based techniques perform poorly. Furthermore, one of the main advantages of P_{abs} over most ways of exploiting landmarks resides in the fact that it is able to account for negative interactions such as delete effects and mutexes. In particular, P_{abs} appears to capture the structure of the problem best when cycles are a key part of the domain. If such negative interactions are not representative of the planning task, using P_{abs} may not be more advantageous than using already existing landmark techniques. Because of this, the performance of any technique based on P_{abs} will depend on the number of landmarks and the features of the domain they represent.

Because of the aforementioned reasons, three different cases arise in problems with meaningful landmarks:

- Almost all the propositions are landmarks: the abstraction is very similar to the original problem and thus it is hard to obtain a balanced trade-off between the information it provides and the difficulty of solving P_{abs}. In this case directly solving P is the simplest alternative, though solving P_{abs} may not be necessarily worse, since the information obtained from P_{abs} might allow solving P with almost no search.

- There are few necessary orders and/or landmark preconditions of the achievers: when solving the abstraction the landmarks do not need to be reachieved, so only a rather arbitrary total order of the landmark graph is obtained. Hence, in most cases the cost of the solution is equal or only slightly higher than h_{LA} with optimal cost partitioning. Besides, partitions of P derived from the obtained total order will not offer much more information than the regular landmark partitioning using disjunctive subgoals.

- There are fairly numerous necessary orders and/or landmark preconditions of the achievers: in these cases an informative plan is obtained with a cost higher than h_{LA} while still being solvable. As described before this is often the case when resources or similarly “consumed” landmarks enforce cycles in the landmark graph, cases in which other techniques often do not fare that well.

Knowing this, it may be relatively simple to predict whether P_{abs} will be useful in the actual resolution of the planning task or not. This work is mainly theoretical, and hence P_{abs} was not used to the efficiency of a given planner. Nevertheless, in order to assess its usefulness in practice, some experimentation has been done. First, P_{abs} was generated in a broad range of domains from the International Planning Competition so the relative size of P_{abs} compared to P could be estimated. The planner used to ground the instances in both cases was Fast Downward (Helmert, 2006), and the chosen measure of size is the number of actions and fluents of each instance after preprocessing, that is, the cardinalities of A, A_{abs}, S and S_{abs} in every instance. The reason why the number of actions was chosen is because it gives an idea of the size of the tasks and because generating P_{abs} often requires action splitting, whose impact can be measured by counting the final number of actions. The number of propositions $|S_{abs}|$ can be known a priori just by computing the landmarks of P, although a significant subset of the propositions derived from the landmarks of P may be static in P_{abs}, which means that $|S_{abs}|$ is often smaller than twice the number of landmarks. Table 8.1 shows the sum of the cardinalities of A, A_{abs}, S and S_{abs} of every instance in each domain. All the orders were used and dominance between actions was enabled. Only propositional landmarks were used, since disjunctive landmarks may lead to an exponential increase in the number of actions.

Results show that the relative cardinality of A_{abs} varies greatly from domain to domain. On one hand, in *Pegsol* $|A_{abs}|$ has the same value as $|A|$; on the other, in some transportation domains like *Transport* or *Zenotravel* $|A_{abs}|$ is comparatively rather small because of the small number of single proposition landmarks found in those domains. This is confirmed by the small value of S_{abs} in those domains, in which $|A_{abs}|$ is significantly smaller than $|A|$. There are no domains in which $|A_{abs}| > |A|$, although in two domains (*Miconic* or *Openstacks*) $|S_{abs}| > |S|$ because of the additional propositions added to represent when a landmark has been achieved.

Additionally, the geometric mean of the ratio between h^*_{abs} and h^{lmcut}/h^{opt}_{LA}/h^{uni}_{LA} is shown for every domain. Only instances in which P_{abs} could be solved optimally under a time limit of 300 seconds were included. A ratio between h^*_{abs} and h^{opt}_{LA} close to 1.00 means that using P_{abs} probably has little potential in those domains; a higher ratio, as in *Barman*, *Blocks*, *Pegsol*, *Pipesworld* (both versions) and *PSR-small* means that using P_{abs} may be promising. Also, note that disjunctive landmarks were not used; if disjunctive landmarks had been used, the ratio could have been higher as well in domains with symmetric resources, like *Gripper* and *Logistics*. The comparison with h^{lmcut} shows that although h^{lmcut} is on average closer to h^*, it varies substantially from domain to domain.

Domain	$\|A\|$	$\|A_{abs}\|$	$\|S\|$	$\|S_{abs}\|$	h^{lmcut}	h_{LA}^{opt}	h_{LA}^{uni}
Airport (50)	144963	125067	157592	46706	0.92	1.00	1.00
Barman-opt11 (20)	13264	11004	4604	1200	0.64	1.97	2.50
Blocks (35)	7490	7434	4826	2549	1.44	1.44	1.44
Depot (22)	68894	51392	9423	1934	0.46	1.00	1.00
Driverlog (20)	53494	5926	6007	542	0.46	1.00	1.00
Elevators-opt08 (30)	18520	7574	3360	607	0.51	1.07	1.07
Elevators-opt11 (20)	11450	4619	2097	571	0.50	1.09	1.09
Floortile-opt11 (20)	9188	6036	3578	1008	0.76	1.06	1.06
Freecell (80)	1071066	663857	23419	13286	2.42	1.03	1.03
Grid (5)	38808	18010	3373	185	0.88	1.01	1.01
Gripper (20)	3720	1880	2380	960	0.52	1.00	1.00
Logistics00 (28)	6972	2380	3429	2409	1.09	1.09	1.09
Logistics98 (35)	501186	13491	82687	3203	1.09	1.09	1.09
Miconic (150)	189100	125128	13950	19964	1.03	1.03	1.03
Mprime (35)	567960	2977	17796	139	0.32	1.00	1.00
Mystery (30)	217800	3634	13066	164	0.36	1.01	1.01
Nomystery-opt11 (20)	72522	59865	4434	1008	1.15	1.09	1.09
Openstacks (30)	213470	212544	10634	11593	1.69	1.00	1.00
Parcprinter-opt08 (30)	9066	2933	6139	4159	0.91	1.00	1.00
Parcprinter-opt11 (20)	5096	1732	3993	2591	0.91	1.00	1.00
Pathways-noneg (30)	40595	7126	13119	2596	0.40	1.00	1.00
Pegsol-opt08 (30)	5346	5346	2920	2003	1.00	1.41	1.41
Pipesworld-notankage (50)	187388	73935	44594	1649	1.03	1.21	1.21
Pipesworld-tankage (50)	1135917	742542	28027	1649	1.06	1.20	1.20
PSR-small (50)	14546	11751	2158	572	1.65	1.65	1.65
Rovers (40)	231653	45064	29324	2705	0.51	1.04	1.04
Satellite (36)	3709130	34163	30479	6192	0.55	1.01	1.01
Scanalyzer-opt08 (30)	1145836	733474	4680	1691	1.05	1.07	1.23
Scanalyzer-opt11 (20)	631288	420820	3088	1010	1.05	1.07	1.18
Sokoban-opt08 (30)	12674	5657	8518	1129	0.59	1.05	1.09
Sokoban-opt11 (20)	7166	3332	5306	706	0.56	1.08	1.11
Tidybot-opt11 (20)	384018	114737	11476	662	0.59	1.09	1.09
Tpp (30)	281351	59833	18807	1564	0.57	1.02	1.02
Transport-opt08 (30)	105888	4200	6800	390	0.02	1.00	1.00
Transport-opt11 (20)	35216	2360	2886	250	0.02	1.00	1.00
Trucks (30)	442262	33896	6964	2065	0.76	1.04	1.04
Visitall-opt11 (20)	3520	2688	2516	2259	1.25	1.00	1.00
Woodworking-opt08 (30)	27835	22058	5677	2729	0.87	1.02	1.03
Woodworking-opt11 (20)	18175	14267	3805	1837	0.90	1.03	1.04
Zenotravel (20)	140433	9138	4518	428	0.45	1.01	1.01

Table 8.1: Comparison between P and P_{abs}: task size and heuristics.

8.7 Related Work

Using landmarks in factored planning (Hoffmann et al., 2004) was the first attempt to exploit the information contained in the landmark graph. However, it did not consider the fact that landmarks can be deleted nor the negative interactions between them. A subsequent work that employed mutexes to build layers of conjunctive landmarks addressed the latter (Keyder et al., 2010). Although more informed than the original partitioning in disjunctive subgoals, computing the new layer was sometimes too inefficient, apart from offering no insight in terms of the formal properties of the problem.

Another work exploited inconsistencies between landmarks and local interactions between achievers and other landmarks to compute the minimum number of states required to satisfy given sets of landmarks (Porteous and Cresswell, 2002), which in turn could be used to obtain a lower bound on the number of times a landmark must be achieved. This information captures the fact that in a sequential plan landmarks may have to be deleted even if they are needed again later and can be used in a cost-partition scheme to derive admissible heuristics with properties similar to the ones mentioned in this work. However, this approach is difficult to define formally due to its procedural nature and offers potentially less information than the generalization of the landmark graph.

Finally, encoding achieved landmarks as goals to enrich abstractions was also proposed (Domshlak et al., 2012). The scope of application of this work was more limited though, and a projection of the enriched problem is less informative than P_{abs} because it does not consider orders nor *e-deletion*.

8.8 Conclusions and Future Work

In this work a generalization of the landmark graph as a planning problem has been presented. So far, the main contribution of the discussed approach is a theoretical one, although several ways of exploiting the obtained abstraction have been proposed. This generalization bridges the gap towards the integration of landmarks in a common framework along with abstractions and heuristics in automated planning. Furthermore, the formal properties that relate it to the original problem have been analyzed.

It remains an open question whether there are additional ways of exploiting or improving the abstraction. On the one hand, it is unclear whether some characteristics of the abstraction, like symmetries or solvability features, can be extrapolated to the original problem. On the other hand, the costs encoded in the generalization could be improved by using cost-partitioning schemes such as the ones used in additive admissible heuristics. For example, the costs of the actions could be substituted by lower bounds on the cost of achieving a given landmark. This is actually already possible in a simple way: if, through the path to a landmark, there are no positive interactions, the cost of the achieved landmark can be substituted by the cost of the path from the closest landmark to the achieved landmark. This is trivial when using actions whose preconditions and effects affect a single invariant, like the *move* operator in the example presented in this work. In this case, the cost of the *move* actions can be the cost of getting to the achieved landmark from the closest landmark, which interestingly enough would make that $h^*_{abs} = h^*$. Other approaches could extend to more general cases, so this line of research may be an interesting follow up of this work.

Part VI

Conclusions

In this thesis we have successfully analyzed the main challenges presented in the introduction and in the objectives. First, we have studied the current state of regression in planning and addressed some of its major drawbacks. In particular, we have focused on applying modern techniques developed for progression to regression. Moreover, taking advantage of the better understanding about classical planning that we have, we also gave some theoretical insights about aspects that were not clear in regression, like reasonable orders and the use of caching schemes for a broad range of reachability heuristics (among which there are very recent developments, like semi-relaxed heuristics). This part of the thesis has lead to the implementation of FDr, a backward search planner more efficient than its predecessor HSPr. This opens the possibility of employing backward search planners to create a potentially very efficient planner, more likely as part of a portfolio or by integrating backward search in a bidirectional planner.

Additionally, the improvement obtained by the use of the proposed techniques has been carried over to symbolic search. We have shown how to exploit constraints popular in explicit-state search in a symbolic setting. Apart from the theoretical significance of this fact, constraints in symbolic search have been used to develop a new generation of symbolic planners whose results are beyond the state of the art in optimal planning. This is of uttermost importance, as until recently symbolic regression was not considered a competitive paradigm by most members of the community.

Following the main motivation of the thesis, a novel way of employing regression along with relaxed plans has allowed us to develop a subgoal generating technique, BGG, that can be exploited in several ways, *i.e.* as a front-to-front forward heuristic. This has been possible thanks to the study of the conditions in which regression is viable.

After proving that regression was an effective way of generating intermediate goals, we studied the complex problem of sampling in implicit search spaces. The use of the states invariants relevant in regression is key, as otherwise the odds of sampling a spurious state are very high in most domains. By formulating the sampling problem as a CSP, the constraints obtained from the states invariants allow planners to avoid spurious states in most of the domains of the current planning benchmarks. The implementation of an algorithm that accurately samples at random the search space opens the possibility of employing a broad range of stochastic algorithms. In particular, Rapidly-exploring Random Trees seemed to have potential, so we have implemented them as part of a satisficing planner, RPT. After describing the implementation details, we performed some experimentation with RPT that shows that it is very competitive with the state of the art. Notably, it surpasses in some domains the performance of LAMA, a very efficient forward search planners that uses a broad range of enhancing techniques.

Finally, we used the knowledge derived from studying both regression and random sampling in the context of landmarks. The main contribution was to exploit invariants to obtain more informed versions of the landmark graph. The first is a SAT compilation of the landmark graph, akin to a scheduling problem that models time steps and state invariants as constraints. The SAT compilation gives not only a guess of the time steps at which a landmark may be necessary (which may used to derive conjunctive intermediate goals and a total order of the landmarks), but also provides sound information of theoretical relevance, like a lower bound on the number of parallel time steps required to solve the problem.

The other alternative characterization of the landmark graph is its formulation as an abstraction of the original problem. This abstraction encodes more information than a mere projection of the original problem and establishes interesting links between the landmark graph and some landmark-based heuristics. This abstraction can also be employed to compute a finer total order of the landmarks or as an admissible heuristic estimate.

The published papers that constitute the core of this thesis are the following:

- "Using Backwards Generated Goals for Heuristic Planning" (Alcázar et al., 2010)
- "A SAT Compilation of the Landmark Graph" (Alcázar and Veloso, 2011)
- "Adapting a Rapidly-Exploring Random Tree for Automated Planning" (Alcázar et al., 2011)
- "Generalization of the Landmark Graph as a Planning Problem" (Alcázar, 2013)
- "Constrained Symbolic Search: On Mutexes, BDD Minimization and More" (Torralba and Alcázar, 2013)
- "Revisiting Regression in Planning" (Alcázar et al., 2013)
- "Automated Context-aware Composition of Advanced Telecom Services for Environmental Early Warnings" (Ordonez et al., 2014)

Bibliography

Alcázar, V. (2013). Generalization of the landmark graph as a planning problem. In *Heuristics and Search for Domain-independent Planning; Workshop at the 23rd International Conference on Automated Planning and Scheduling*, pages 175–183.

Alcázar, V., Borrajo, D., Fernández, S., and Fuentetaja, R. (2013). Revisiting regression in planning. In *International Joint Conference on Artificial Intelligence*, pages 2254–2260.

Alcázar, V., Borrajo, D., and Linares López, C. (2010). Using backwards generated goals for heuristic planning. In *International Conference on Automated Planning and Scheduling*, pages 2–9.

Alcázar, V. and Veloso, M. (2011). A SAT compilation of the landmark graph. In *COPLAS*, pages 47–54.

Alcázar, V., Veloso, M. M., and Borrajo, D. (2011). Adapting a rapidly-exploring random tree for automated planning. In *Symposium on Combinatorial Search (SoCS)*, pages 2–9.

Aurenhammer, F. (1991). Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405.

Bäckström, C. and Nebel, B. (1995). Complexity results for SAS+ planning. *Computational Intelligence*, 11:625–656.

Bahar, R. I., Frohm, E. A., Gaona, C. M., Hachtel, G. D., Macii, E., Pardo, A., and Somenzi, F. (1997). Algebraic decision diagrams and their applications. *Formal Methods in System Design*, 10(2/3):171–206.

Baier, J. A. and Botea, A. (2009). Improving planning performance using low-conflict relaxed plans. In *International Conference on Automated Planning and Scheduling*, pages 10–17.

Bibai, J., Savéant, P., Schoenauer, M., and Vidal, V. (2010). An evolutionary metaheuristic based on state decomposition for domain-independent satisficing planning. In *International Conference on Automated Planning and Scheduling*, pages 18–25. AAAI.

Blum, A. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300.

Bonet, B. (2013). An admissible heuristic for SAS+ planning obtained from the state equation. In *International Joint Conference on Artificial Intelligence*, pages 2268–2274.

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

Borrajo, D. and Veloso, M. (2012). Probabilistically reusing plans in deterministic planning. In *Proceedings of ICAPS'12 workshop on Heuristics and Search for Domain-Independent Planning*, pages 17–25, Atibaia (Brazil).

Botea, A., Enzenberger, M., Müller, M., and Schaeffer, J. (2005). Macro-FF: Improving ai planning with automatically learned macro-operators. *J. Artif. Intell. Res. (JAIR)*, 24:581–621.

Brafman, R. I. (2006). Factored planning: How, when, and when not. In *In Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-2006*, pages 809–814.

Brélaz, D. (1979). New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256.

Bruce, J. and Veloso, M. M. (2006). Real-time randomized motion planning for multiple domains. In Lakemeyer, G., Sklar, E., Sorrenti, D. G., and Takahashi, T., editors, *RoboCup*, volume 4434 of *Lecture Notes in Computer Science*, pages 532–539.

Bryant, R. E. (1986). Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691.

Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318.

Burfoot, D., Pineau, J., and Dudek, G. (2006). RRT-Plan: A randomized algorithm for STRIPS planning. In *International Conference on Automated Planning and Scheduling*, pages 362–365.

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

Cai, D., Hoffmann, J., and Helmert, M. (2009). Enhancing the context-enhanced additive heuristic with precedence constraints. In *International Conference on Automated Planning and Scheduling*, pages 50–57.

Chen, Y., Xing, Z., and Zhang, W. (2007). Long-distance mutual exclusion for propositional planning. In *International Joint Conference on Artificial Intelligence*, pages 1840–1845.

Christie, M., Machap, R., Normand, J.-M., Olivier, P., and Pickering, J. (2005). Virtual camera planning: A survey. In Butz, A., Fisher, B., KrÃ¼ger, A., and Olivier, P., editors, *Smart Graphics*, volume 3638 of *Lecture Notes in Computer Science*, pages 40–52. Springer Berlin Heidelberg.

Coles, A., Fox, M., Long, D., and Smith, A. (2008). Teaching forward-chaining planning with JavaFF. In *Colloquium on AI Education, Twenty-Third AAAI Conference on Artificial Intelligence*.

Cortés, J., Siméon, T., De Angulo, V. R., Guieysse, D., Remaud-Siméon, M., and Tran, V. (2005). A path planning approach for computing large-amplitude motions of flexible molecules. *Bioinformatics*, 21(suppl 1):i116–i125.

Coudert, O. and Madre, J. C. (1990). A unified framework for the formal verification of sequential circuits. In *International Conference on Computer-Aided Design (ICCAD)*, pages 126–129.

Culberson, J. C. and Schaeffer, J. (1998). Pattern databases. *Comput. Intell.*, 14(3):318–334.

Da Xu, L., Wang, C., Bi, Z., and Yu, J. (2012). Autoassem: an automated assembly planning system for complex products. *Industrial Informatics, IEEE Transactions on*, 8(3):669–678.

Dillenburg, J. F. and Nelson, P. C. (1994). Perimeter search. *Artificial Intelligence*, 65(1):165–178.

Domshlak, C., Katz, M., and Lefler, S. (2012). Landmark-enhanced abstraction heuristics. *Artificial Intelligence*, 189:48–68.

Edelkamp, S. (2002). Symbolic pattern databases in heuristic search planning. In *Conference on Artificial Intelligence Planning Systems (AIPS)*, pages 274–283.

Edelkamp, S. and Helmert, M. (2000). On the implementation of MIPS. In *In Proceedings of Workshop on DecisionTheoretic Planning, Artificial Intelligence Planning and Scheduling (AIPS*.

Edelkamp, S. and Kissmann, P. (2011). On the complexity of BDDs for state space search: A case study in connect four. In *AAAI Conference on Artificial Intelligence*.

Edelkamp, S. and Reffel, F. (1998). OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, pages 81–92.

Fikes, R. and Nilsson, N. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. In *Proceedings of the 2nd international joint conference on Artificial intelligence*, pages 608–620, San Francisco, CA, USA.

Fox, M. and Long, D. (2003). Pddl2.1: An extension to pddl for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124.

Fuentetaja, R., Borrajo, D., and Linares López, C. (2009). A unified view of cost-based heuristics. In *Workshop on Heuristics for Domain-Independent Planning, ICAPS'09*, Thessaloniki (Greece).

Gerevini, A., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668.

Gerevini, A. and Serina, I. (2002). LPG: A planner based on local search for planning graphs with action costs. In *Conference on Artificial Intelligence Planning Systems*, pages 13–22. AAAI.

Goerzen, C., Kong, Z., and Mettler, B. (2010). A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57(1-4):65–100.

Haralick, R. M. and Elliott, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313.

Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107.

Haslum, P. (2008). Additive and reversed relaxed reachability heuristics revisited. *Proceedings of the 6th International Planning Competition.*

Haslum, P. (2009). $h^m(P) = h^1(P^m)$: Alternative characterisations of the generalisation from $h^{\max}$ to h^m. In *International Conference on Automated Planning and Scheduling*, pages 354–357.

Haslum, P., Botea, A., Helmert, M., Bonet, B., and Koenig, S. (2007). Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 1007–1012.

Helmert, M. (2004). A planning heuristic based on causal graph analysis. In *International Conference on Automated Planning and Scheduling*, pages 161–170.

Helmert, M. (2006). The Fast Downward planning system. *J. Artif. Intell. Res. (JAIR)*, 26:191–246.

Helmert, M. (2009). Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503–535.

Helmert, M. and Domshlak, C. (2009). Landmarks, critical paths and abstractions: What's the difference anyway? In *International Conference on Automated Planning and Scheduling*, pages 162–169.

Helmert, M. and Geffner, H. (2008). Unifying the causal graph and additive heuristics. In *International Conference on Automated Planning and Scheduling*, pages 140–147.

Helmert, M., Röger, G., and Karpas, E. (2011). Fast Downward Stone Soup: A baseline for building planner portfolios. In *Workshop on Planning and Learning (PAL)*, pages 28–35.

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

Hoffmann, J., Porteous, J., and Sebastia, L. (2004). Ordered landmarks in planning. *J. Artif. Intell. Res. (JAIR)*, 22:215–278.

Holte, R. C., Perez, M. B., Zimmer, R. M., and MacDonald, A. J. (1996). Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI, Vol. 1*, pages 530–535.

Hong, Y., Beerel, P. A., Burch, J. R., and McMillan, K. L. (1997). Safe BDD minimization using don't cares. In *In Design Automation Conference (DAC)*, pages 208–213.

Hong, Y., Beerel, P. A., Burch, J. R., and McMillan, K. L. (2000). Sibling-substitution-based BDD minimization using don't cares. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 19(1):44–55.

Jensen, R. M., Hansen, E. A., Richards, S., and Zhou, R. (2006). Memory-efficient symbolic heuristic search. In *International Conference on Automated Planning and Scheduling*, pages 304–313.

Karpas, E. and Domshlak, C. (2009). Cost-optimal planning with landmarks. In *International Joint Conference on Artificial Intelligence*, pages 1728–1733.

Kautz, H. A. and Selman, B. (1999). Unifying SAT-based and graph-based planning. In *International Joint Conference on Artificial Intelligence*, pages 318–325.

Kautz, H. A., Selman, B., and Hoffmann, J. (2006). SatPlan: Planning as satisfiability. In *Abstracts of the 5th International Planning Competition*.

Kavraki, L., Svestka, P., Kavraki, L., Latombe, J., and Overmars, M. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12:566–580.

Keyder, E. and Geffner, H. (2008). Heuristics for planning with action costs revisited. In *European Conference on Artificial Intelligence*, pages 588–592.

Keyder, E. and Geffner, H. (2009a). Soft goals can be compiled away. *J. Artif. Intell. Res. (JAIR)*, 36:547–556.

Keyder, E. and Geffner, H. (2009b). Trees of shortest paths vs. steiner trees: Understanding and improving delete relaxation heuristics. In *International Joint Conference on Artificial Intelligence*, pages 1734–1739.

Keyder, E., Hoffmann, J., and Haslum, P. (2012). Semi-relaxed plan heuristics. In *International Conference on Automated Planning and Scheduling*, pages 128–136.

Keyder, E., Richter, S., and Helmert, M. (2010). Sound and complete landmarks for and/or graphs. In *ECAI*, pages 335–340.

Kissmann, P. and Edelkamp, S. (2011). Improving cost-optimal domain-independent symbolic planning. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 992–997.

Kissmann, P. and Hoffmann, J. (2013). What's in it for my BDD? On causal graphs and variable orders in planning. In *International Conference on Automated Planning and Scheduling*, pages 327–331.

Koehler, J. and Hoffmann, J. (2000). On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *J. Artif. Intell. Res. (JAIR)*, 12:338–386.

Kuffner, J. J. and LaValle, S. M. (2000). RRT-Connect: An efficient approach to single-query path planning. In *ICRA*, pages 995–1001. IEEE.

LaValle, S. M. and Kuffner, J. J. (1999). Randomized kinodynamic planning. In *International Conference on Robotics and Automation*, pages 473–479.

Linares López, C. and Borrajo, D. (2010). Adding diversity to classical heuristic planning. In *Proceedings of the Third Annual Symposium on Combinatorial Search (SOCS'10)*, pages 73–80, Atlanta, USA.

Lipovetzky, N. and Geffner, H. (2011). Searching for plans with carefully designed probes. In *International Conference on Automated Planning and Scheduling*, pages 154–161.

Lipovetzky, N. and Geffner, H. (2012). Width and serialization of classical planning problems. In *European Conference on Artificial Intelligence*, pages 540–545.

Liu, Y., Koenig, S., and Furcy, D. (2002). Speeding up the calculation of heuristics for heuristic search-based planning. In *AAAI Conference on Artificial Intelligence*, pages 484–491, Menlo Park, CA, USA. American Association for Artificial Intelligence.

Lu, Q., Xu, Y., Huang, R., and Chen, Y. (2011). The Roamer planner random-walk assisted best-first search. In *The 2011 International Planning Competition*.

Marzal, E., Sebastia, L., and Onaindia, E. (2008). Detection of unsolvable temporal planning problems through the use of landmarks. In *European Conference on Artificial Intelligence*, pages 919–920.

Massey, B. (1999). *Directions In Planning: Understanding The Flow Of Time In Planning*. PhD thesis, Computational Intelligence Research Laboratory, University of Oregon.

Mccarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, pages 463–502.

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, pages 463–502.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

McMillan, K. L. (1993). *Symbolic model checking*. Kluwer Academic publishers.

McMillan, K. L. (1996). A conjunctively decomposed boolean representation for symbolic model checking. In *Computer Aided Verification (CAV)*, pages 13–25.

Morgan, S. and Branicky, M. S. (2004). Sampling-based planning for discrete spaces. In *International Conference on Robotics and Automation*.

Nakhost, H. and Müller, M. (2009). Monte-Carlo exploration for deterministic planning. In *International Joint Conference on Artificial Intelligence*, pages 1766–1771.

Newell, A., Shaw, J. C., and Simon, H. A. (1959). Report on a general problem-solving program. In *International Conference on Information Processing*, pages 256–264.

Nguyen, X. and Kambhampati, S. (2001). Reviving partial order planning. In *International Joint Conference on Artificial Intelligence*, pages 459–466.

Nigenda, R. S., Nguyen, X., and Kambhampati, S. (2000). Altalt: Combining the advantages of graphplan and heuristic state search. *Knowledge Based Computer Systems*, pages 409–421.

Ordonez, A., Alcázar, V., Corrales, J. C., and Falcarini, P. (2014). Automated context aware composition of advanced telecom services for environmental early warnings. *Expert Systems with Applications*, 41(13):5907 – 5916.

Pang, B. and Holte, R. C. (2011). State-set search. In *Symposium on Combinatorial Search*, pages 125–133.

Pednault, E. P. D. (1989). Adl: exploring the middle ground between strips and the situation calculus. In *Principles of Knowledge Representation and Reasoning*, pages 324–332.

Penberthy, J. S. and Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *Principles of Knowledge Representation and Reasoning*, pages 103–114.

Pettersson, M. P. (2005). Reversed planning graphs for relevance heuristics in ai planning. *Frontiers in AI and Applications*, pages 29–38.

Porteous, J. and Cresswell, S. (2002). Extending landmarks analysis to reason about resources and repetition. In *PLANSIG*.

Refanidis, I. and Vlahavas, I. (2004). The GRT planner: Backward heuristic construction in forward state-space planning. *J. Artif. Intell. Res. (JAIR)*, 22:215–278.

Reiter, R. (1977). On closed world data bases. In *Logic and Data Bases*, pages 55–76.

Richter, S. (2010). *Landmark-Based Heuristics and Search Control for Automated Planning*. PhD thesis, Griffith University.

Richter, S. and Helmert, M. (2009). Preferred operators and deferred evaluation in satisficing planning. In *International Conference on Automated Planning and Scheduling*, pages 273–280.

Richter, S., Thayer, J. T., and Ruml, W. (2010). The joy of forgetting: Faster anytime search via restarting. In Brafman, R. I., Geffner, H., Hoffmann, J., and Kautz, H. A., editors, *International Conference on Automated Planning and Scheduling*, pages 137–144.

Richter, S. and Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res. (JAIR)*, 39:127–177.

Rintanen, J. (2008). Regression for classical and nondeterministic planning. In *European Conference on Artificial Intelligence*, pages 568–571.

Rintanen, J. (2010). Heuristics for planning with SAT. In *Principles and Practice of Constraint Programming (CP)*, pages 414–428.

Rivest, R. L. (1974). *Analysis of Associative Retrieval Algorithms.* PhD thesis, Stanford, CA, USA.

Röger, G. and Helmert, M. (2010). The more, the merrier: Combining heuristic estimators for satisficing planning. In *International Conference on Automated Planning and Scheduling*, pages 246–249.

Sebastia, L., Marzal, E., and Onaindia, E. (2007). Extracting landmarks in temporal planning domains. In *International Conference on Artificial Intelligence (IC-AI'07)*, pages 520–526.

Sebastia, L., Onaindia, E., and Marzal, E. (2006). Decomposition of planning problems. *AI Commun.*, 19(1):49–81.

Torralba, Á. and Alcázar, V. (2013). Constrained symbolic search: On mutexes, BDD minimization and more. In *Symposium on Combinatorial Search (SoCS)*, pages 175–183.

Torralba, Á., Edelkamp, S., and Kissmann, P. (2013a). Transition trees for cost-optimal symbolic planning. In *International Conference on Automated Planning and Scheduling*.

Torralba, Á., Linares López, C., and Borrajo, D. (2013b). Symbolic merge-and-shrink for cost-optimal planning. In *International Joint Conference on Artificial Intelligence*.

Veloso, M. M., Carbonell, J. G., Pérez, M. A., Borrajo, D., Fink, E., and Blythe, J. (1995). Integrating planning and learning: the prodigy architecture. *J. Exp. Theor. Artif. Intell.*, 7(1):81–120.

Vidal, V. (2004a). A lookahead strategy for heuristic search planning. In *International Conference on Automated Planning and Scheduling*, pages 150–159.

Vidal, V. (2004b). A lookahead strategy for heuristic search planning. In Zilberstein, S., Koehler, J., and Koenig, S., editors, *International Conference on Automated Planning and Scheduling*, pages 150–160.

Vidal, V. and Geffner, H. (2005). Solving simple planning problems with more inference and no search. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, volume 3709 of *LNCS*, pages 682–696, Sitges, Spain. Springer.

Xie, F., Nakhost, H., and Müller, M. (2012). Planning via random walk-driven local search. In *International Conference on Automated Planning and Scheduling*.

Xing, Z., Chen, Y., and Zhang, W. (2006). Optimal STRIPS planning by maximum satisfiability and accumulative learning. In *International Conference on Automated Planning and Scheduling*, pages 442–446.

Younes, H. L. and Littman, M. L. (2004). PPDDL1.0: The language for the probabilistic part of IPC-4. In *Proceedings of the International Planning Competition*, page 46.

Younes, H. L. S. and Simmons, R. G. (2003). VHPOP: Versatile heuristic partial order planner. *J. Artif. Intell. Res. (JAIR)*, 20:405–430.

Zhu, L. and Givan, R. (2003). Landmark extraction via planning graph propagation. In *Doctoral Consortium of the International Conference on Automated Planning and Scheduling*.

Zilles, S. and Holte, R. C. (2010). The computational complexity of avoiding spurious states in state space abstraction. *Artificial Intelligence*, 174(14):1072–1092.