DEPARTMENT OF COMPUTER SCIENCE

Andreas Sideris

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

February, 2015

APPROVAL PAGE

Doctor of Philosophy Dissertation

ADVANCES IN SAT-BASED PLANNING

Presented by

Andreas Sideris

Research Supervisor	Yannis Dimopoulos
Committee Member	Antonis Kakas
Committee Member	Carmel Domshlak
Committee Member	Hector Geffner
Committee Member	Elpida Keravnou Papailiou

University of Cyprus

February, 2015

DECLARATION OF DOCTORAL CANDIDATE

The present doctoral dissertation was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy of the University of Cyprus. It is a product of original work of my own, unless otherwise mentioned through references, notes, or any other statements.

Andreas Sideris

....

ΕΞΕΛΙΞΕΙΣ ΣΤΟ ΣΧΕΔΙΑΣΜΟ ΔΡΑΣΗΣ ΒΑΣΙΖΟΜΕΝΟ ΣΤΗ ΠΡΟΤΑΣΙΑΚΗ ΛΟΓΙΚΗ

Ο σχεδιασμός δράσης είναι ένα δύσκολο πρόβλημα. Ακόμα και στις πιο απλές του μορφές είναι υπολογιστικά δυσεπίλυτες ('intractable'). Παρόλο που είναι απίθανος ο αποτελεσματικός (ως προς χρόνο) σχεδιασμός δράσης στη γενική περίπτωση, εντούτοις αποτελεσματικές ευρετικές μέθοδοι και αποδοτικοί αλγόριθμοι διάχυσης περιορισμών είναι πολύτιμες τεχνικές για την επίλυση μεγάλων προβλημάτων σχεδιασμού δράσης. Πράγματι, πολλά μοντέρνα συστήματα σχεδιασμού δράσης μετατρέπουν το πρόβλημα σχεδιασμού δράσης σε πρόβλημα επίλυσης περιορισμών, όπως π.χ. προτασιακή λογική, και στη συνέχεια το λύνουν με αλγορίθμους προτασιακής λογικής (SAT solvers), περιορισμών ή ψευδό-προτασιακης λογικής ('Pseudo-boolean'). Πολλά άλλα συστήματα λογισμικού επιλύουν το πρόβλημα κατευθύνοντας την αναζήτηση με αποτελεσματικές ευρετικές μεθόδους που εξάγονται αυτόματα από το ίδιο το πρόβλημα.

Στα πλαίσια αυτής της διατριβής υλοποιήσαμε πρώτα το σύστημα SMP, ένα νέο τρόπο κωδικοποίησης των προβλημάτων σχεδιασμού δράσης σε προτασιακή λογική (SAT). Αποδεικνύουμε τόσο θεωρητικά αλλά και πειραματικά ότι οι μηχανισμοί διάχυσης περιορισμών των μοντέρνων επιλυτών προτασιακής λογικής διαχέουν τους περιορισμούς πολύ αποδοτικότερα στο SMP από άλλους τρόπους κωδικοποίησης. Επιπλέον, με τη χρήση λογισμικού που υλοποιήσαμε, βρίσκουμε επιπρόσθετους δυαδικούς περιορισμούς που ισχύουν σε προβλήματα σχεδιασμού δράσης για τους οποίους παραθέτουμε ισχυρές πειρατικές ενδείξεις η προσθήκη τους στο SMP δεν προσφέρει υπολογιστικά οφέλη. Στη συνέχεια η κωδικοποίηση του SMP χρησιμοποιήθηκε σαν βάση στην ανάπτυξη του

συστήματος σχεδιασμού δράσης PSP. Το σύστημα PSP μεγιστοποιεί το πλήθος των στόχων (goals) που επιτυγχάνονται σε ένα περιορισμένο χρονικό ορίζοντα, επαναλαμβάνοντας τη διαδικασία για διαδοχικά μεγαλύτερους ορίζοντες. Παρά την αδυναμία του PSP να εγγυηθεί ότι οι λύσεις είναι βέλτιστές (ως προς το μήκος τους) όπως το σύστημα SMP, εντούτοις υπολογίζει σχέδια δράσης καλής ποιότητας για προβλήματα που δεν μπορούν να επιλυθούν από τον SMP ή άλλα συστήματα σχεδιασμού δράσης (που παράγουν βέλτιστες λύσεις) τα οποία βασίζονται σε προτασιακούς επιλυτές.

Ένα βασικό μειονέκτημα του συστήματος PSP είναι η περιορισμένη του αποδοτικότητα σε προβλήματα μεγάλου μεγέθους. Αυτό οφείλεται στο ότι οι κωδικοποιήσεις μεγάλων προβλημάτων σχεδιασμού δράσης είναι συχνά πολύ δύσκολες για τους προτασιακούς επιλυτές. Αυτό ισχύει για όλα τα συστήματα σχεδιασμού δράσης που βασίζονται στην μέθοδο της διαδοχικής επέκτασης του ορίζοντα δράσης, αφού το μέγεθος των κωδικοποιήσεων μεγαλώνει όσο μεγαλώνει και ο ορίζοντας. Η αδυναμία αυτή αντιμετωπίζεται από το σύστημα PSP-H το οποίο επεκτείνει το σύστημα PSP με δύο αποδοτικές τεχνικές που αποσυνθέτουν το πρόβλημα σε μικρότερα υποπροβλήματα ώστε τα προβλήματα προτασιακής ικανοποιησιμότητας που προκύπτουν να μην είναι υπερβολικά μεγάλα. Η πρώτη τεχνική μετατρέπει το πρόβλημα σχεδιασμού δράσης σε μια σειρά προβλημάτων δυαδικής βελτιστοποίησης, σε κάθε ένα από τα οποία ο στόχος είναι να μεγιστοποιηθεί το πλήθος των στόχων (goals) που μπορούν να ικανοποιηθούν εντός ενός περιορισμένου ορίζοντα (μήκος πλάνου). Αυτή η τεχνική συνδυάζεται με μια δεύτερη τεχνική που κατευθύνει την αναζήτηση για κάθε υποπρόβλημα σε μια κατάσταση όπου ικανοποιούνται όλοι οι στόχοι του αρχικού προβλήματος. Τα πειραματικά μας αποτελέσματα αποδεικνύουν ότι το PSP-H είναι ένας ανταγωνιστικός αλγόριθμος σχεδιασμού δράσης .

ADVANCES IN SAT-BASED PLANNING

Planning is a difficult problem. Even in its simplest forms it is computationally intractable. Although it is unlikely to be able to plan efficiently in the general case, good heuristics and strong constraint propagation methods are valuable techniques for tackling large planning problems. Indeed, some modern planners transform planning into a constraint satisfaction problem, such as a boolean formula, and then solve it by invoking a satisfiability, constraint or pseudo-boolean solver. Many other planners solve the problem by guiding the search using powerful heuristics that are automatically extracted from the planning domain.

In the context of this work we first implemented `SMP`, a novel way of transforming a planning domain into a propositional boolean formula (SAT). We prove both theoretically and experimentally that the constraint propagation engines of the modern SAT solvers propagate the constraints much more efficiently in `SMP` than in previous transformations. We also provide strong experimental evidence that the addition of more implied non-redundant binary constraints to `SMP` does not improve the planning times. We then use the SAT encoding of `SMP` in the `PSP` planner. `PSP` seeks to maximize the number of goals that can be achieved using the solve and expand method. Although `PSP` cannot guarantee optimality as `SMP`, it often generates sub-optimal plans of high quality for planning problems that are beyond the reach of `SMP` and other optimal SAT-based planners. A drawback of the `PSP` planner is its limited scalability, as the instances that arise from large planning problems are often too hard for SAT solvers. This holds true for all planners based on the solve and expand method, as the size of the SAT instance grows monotonically with the planning horizon. To address this problem we developed the `PSP-H` planning system, that extends `PSP` by combining two powerful techniques that aim at decomposing a planning problem into smaller subproblems, so that the instances that need to be solved do not grow prohibitively large. The first technique turns planning into a series of boolean optimization problems, each seeking to maximize the number of goals that are achieved within a limited planning horizon. This is coupled with a second technique that directs search towards a state that satisfies all goals. Experimental results demonstrate that `PSP-H` is a competitive planning algorithm.

ACKNOWLEDGEMENTS

First of all I want to thank my research advisor, associate professor *Yannis Dimopoulos* for his constant support, guidance and patience for all the (many) years towards the fulfilment of this Ph.D. thesis. After fourteen years of collaboration and tens of thousands of source code lines with *Yannis Dimopoulos*, starting when I was an undergraduate student, I do consider *Yannis* not only an advisor and a mentor but also a very good friend.

I also thank the members of my examination committee professors *Antonis Kakas*, *Elpida Keravnou Papailiou*, *Carmel Domshlak* and *Hector Geffner* for their participation.

I also thank *Stefanos Stylianou* for implementing an early minimal prototype version of the PSP planning system in his final year project.

I would also like to thank my good friend Dr. *Kyriakos Christou* for his valuable help with the latex editor.

I want to thank assistant professor *Loizos Michael* for some fruitful discussions we had regarding this research during the Knowledge Representation and Reasoning 2014 conference in Vienna.

I want to thank my parents *Georgios* and *Salomi* for their constant support and love.

Finally I would like to express my gratitude and love to my wife *Maria* for her patience and support during all these years. She proved that family, full time work, and research are not (totally) incompatible. I also want to apologise to my three children *Giorgos*, *Sotiroula* and *Michalis* for the times I was too busy to spend precious time with them.

DEDICATION

To my wife *Maria* and my three children *Giorgos*, *Sotiroula* and *Michalis*

CREDITS

The publications that lead to this thesis are: [111] [112] [113]

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Planning

A planning problem is the task of coming up with a sequence of actions that achieve a goal [108]. Planning comes in many forms [47, 45]. Finding a plan that will be executed in a deterministic world is quite different than planning in the presence of non-determinism. On the other hand, computing an optimal plan can be more difficult than finding an arbitrary plan. The situation can be further complicated when actions have different durations and incur different costs. Moreover a general-purpose planner for a wide range of domains is likely to be built differently from a system for a specific domain.

One way of dealing with automated planning is to transform it to a Constraint Satisfaction Problem (CSP) and solve it with a general constraint solver. Constraint Satisfaction (CS) is used in practice for solving hard problems arising in many different fields such as robotics, economics, bioinformatics, engineering, scheduling, computer vision, routing and many others. Loosely speaking, the idea of CS is to provide systems where problems from various fields of expertise can be expressed in a simple and intuitive language. The input of the general system is a set of variables, a set of possible values for each of these variables, and a set of relations (constraints) that must simultaneously hold for their values. Although the user does not provide any information on how the problem is to be solved (CS can be seen as a language for *declarative programming*), the *solver* of the CS system computes a solution, i.e. a mapping of variables to values such that no constraint is violated, or reports that no solution exists.

A special case of CSP is SAT, where the variables are propositional variables that can be either true or false, and the constraints are propositional logic formulae that need to be satisfied. One of the approaches to automated planning is to cast a planning problem as a CSP or SAT, and invoke a solver that generates a plan. The way a planning problem is transformed into a CSP (or SAT), is of paramount importance for the success of the approach. Moreover, solving a CSP usually involves a combination of different techniques such as good methods for reasoning with constraints, known as *constraint propagation*, heuristics, and exploitation of problem structure.

On the other hand, state-based heuristic search planning, a different approach to solving planning problems, has seen tremendous progress over the last years, with heuristic planners becoming gradually more effective than CS/SAT based systems. Indeed, many problems that are unsolvable by the best SAT-based planners are easy for heuristic search systems. From a practical perspective, this thesis contributes some ideas towards bridging part of the gap in the effectiveness between SAT-based and heuristic planners.

1.2 Motivation

In this thesis we investigate classical or propositional planning, which is planning in its simplest form. In classical planning the world is finite and deterministic and actions are executed as outlined by the plan (actions cannot fail). The actions have no durations or costs, and the state of the problem is changed only by the actions of the plan in a deterministic way. Moreover the initial state of the problem is fully known. Classical planning may seem 'simple' or 'naive' for the real world. Indeed, it is hard to imagine realistic problems with no uncertainty at all or with actions without costs or durations. However even classical planning is not an easy problem as it is *PSPACE-complete* in general, and *NP-complete* for a fixed planning horizon [26]. Hence, solving classical planning requires sophisticated methods. Moreover, techniques that prove useful in solving classical planning can form the basis for deriving algorithms for more complex problems.

The use of CSP or SAT is a 'standard' method for solving *NP-complete* problems. This is true for planning as well. Indeed, from the early days of automated planning many planners followed the general approach of translating planning into CSP or SAT, that is then given as input to a general-purpose solver. However, despite its intuitive appeal, current implementation of this general idea has serious limitations, that can be addressed by devising stronger propagation methods as well as techniques and heuristics that exploit problem structure. This is exactly the approach that is taken in this thesis. More precisely:

1. We study how SAT models of planning interact with the underlying solver, especially with the constraint propagation engine of the solver and explain differences in their performance. The idea is to propose improved models that exhibit better behaviour due to better interaction with the solver.

2. We study how the structure of a planning problem can be exploited in solving these problems. For instance, the SAT theory corresponding to a planning problem is "layered", where layers correspond to different time points.

3. We investigate ways of incorporating ideas from heuristic search in the SAT models for planning. This seems an obvious direction since the best (sub-optimal) classical planners available use powerful heuristics that are automatically extracted from the problem to guide the search rapidly from the initial state of the problem to a state that satisfies the goals.

1.3 Thesis structure

Chapters 2 and 3 present notations, background knowledge and relevant research in the field, whereas chapters 4,5 and 6 are the core of this Ph.D. thesis and its research contribution. Conclusions and future directions are presented in chapter 7.

Chapter 2 presents preliminaries, definitions and background knowledge on constraint satisfaction and propositional satisfiability.

Chapter 3 introduces notations, background knowledge and the research performed in propositional planning that is relevant to our work. Some related work that is very close to ours is presented instead in the appropriate chapters 4,5 or 6 respectively.

Chapter 4 presents the SMP planning system. SMP employs a new encoding and builds on the SATPLAN framework [76], coupled with the SAT solver *precosat* [16]. We describe in detail the encoding, prove theoretical results that reveal its strength, and provide experimental results that demonstrate its practical value. We also provide experimental results suggesting that implied non-redundant binary constraints that are added in the SAT theory do not improve planning times.

Chapter 5 describes the PSP planning system. PSP uses the SMP translation of the planning problem and extends the planning as propositional satisfiability (the SATPLAN framework) to planning as pseudo-boolean optimization. The approach follows the classic solve and expand method of the SATPLAN framework, but at each step it seeks to maximize the number of the problem goals that can be achieved. The PSP algorithm is described in detail and experimental results are provided.

Chapter 6 is devoted to additional improvements implemented in the PSP-H planning system. PSP-H extends the PSP by combining two powerful techniques that aim at decomposing a planning problem into smaller subproblems. The first technique, *incremental goal achievement*, turns planning into a series of boolean optimization problems, each seeking to maximize the number of goals that are achieved within a limited planning horizon. This is coupled with a second technique, called *heuristic guidance*, that directs search towards a state that satisfies all goals. The `PSP-H` algorithm is described in detail and experimental results are provided.

Conclusions and directions for future research work are presented in chapter 7. We briefly describe some ideas that could further improve the performance of the implemented planners, as well as the quality of the plans.

Chapter 2

Constraint Satisfaction and Propositional Satisfability

2.1 Constraint satisfaction problems and constraint propagation

Constraint Satisfaction (CS) has been applied to solve hard real world problems Constraint Satisfaction Problems (or CSP) that arise in many different fields [107, 122] such as robotics [117], business (nurse scheduling [28] and agriculture [2]), bioinformatics [8], electricity and fluid (water and oil) networks [114] , and vehicle routing [27]. The idea of CS is to render possible for the scientist of the field to input in a CS solver the variables of the system he/she is studying, the possible values of these variables (domains), and relations (constraints) that must hold for the possible values of the variables, and to find a solution: a mapping of variables to values such that no constraint is violated. In this section we give the definitions as they are presented by Christian Bessiere in [107], chapter 3.

Definition 1 A *constraint satisfaction problem* (CSP) P, is a triple $P =< X, D, C >$ where X is an n-tuple of variables $X =< x_1, x_2, \cdots, x_n >$, D is the n-tuple $D =< D_1, D_2, \cdots, D_n >$ of domains corresponding to variables (i.e. $\forall i \in [1 \dots n]$, x_i takes a value from the domain D_i) and C is a τ-tuple of constraints $C =< c_1, c_2, \cdots, c_\tau >$ over the values of the variables. A *constraint*

c_j is a pair $c_j =< R_{S_j}, S_j >$ where R_{S_j} is a relation on the variables of the $S_j \subseteq X$ which is the scope of c, $S_j = scope(c)$. R_{S_j} is a subset of the Cartesian product over the domains D of variables in S_j. The scope (or *scheme*) of a $c_i \in C$ is denoted as $X(c_i)$.

Definition 2 A *constraint satisfaction problem* $P =< X, D, C >$ is a *binary constraint satisfaction problem* if for all $c_i \in C$, $|X(c_i)| = 2$.

An assignment A for a problem $P =< X, D, C >$ is the mapping of some of the variables of X to a value from their respective domains D. The assignment is partial if not all the variables are mapped to a value, and total otherwise. A constraint $c_j \in C$ is satisfied by an assignment A if the projection of A over the scope of c_j is a subset of R_{S_j}. An assignment A of a problem is a solution to the problem P if and only if it is total and all the constraints of the problem are satisfied.

Solving *Constraint Satisfaction Problems* is computationally hard (*NP-complete*) in the general case. A special case of CSP is SAT, where all the variables are propositions, and all the constraints are propositional formulas.

2.1.1 Constraint Propagation Methods

Constraint propagation methods are one of the more - if not the most - central concept in constraint programming. The idea is to infer new constraints by reasoning on the state of the problem already known, reducing the space that remains to be explored in order to find a solution faster. For example assume a CSP $P =< X, D, C >$ such that $x_1, x_2 \in X$, $D_{x_1}, D_{x_2} \in D$, $D_{x_1} = D_{x_2} = [1 \ldots 10]$ and $c_i \in C, c_i = x_1 + x_2 < 5$. If the value 2 is assigned to variable x_1, then any value $k \in D_{x_2}, k \geq 4$ can be correctly removed from the domain D_{x_2} since the constrained c_i would be violated for any value equal or greater than 4 when $x_1 = 2$. Thus D_{x_2} is correctly updated (or filtered) to $D_{x_2} = [1...3]$. This value removal, often called ”domain filtering", is a powerful method for propagating constraints, as it reduces the size of the search space.

Definition 3 A CSP problem $P =< X, D, C >$ is General Arc Consistent (GAC) if and only if all its constraints c in C are General Arc Consistent. A constraint c in C of P is General Arc Consistent (GAC) if each value d of each variable v in $scope(c)$ is consistent with the rest variables $v_1, v_2, \cdots, v_r$ of $scope(c)$. A constraint c with $scope(c) = \{v, v_1, \ldots, v_r\}$ is consistent if there exist values $d, d_1, \ldots d_r$, $d \in D_v, d_i \in D_{x_i} \forall i \in [1 \ldots r]$ such that the assignment (or tuple) $(v, d), (v_1, d_1), (v_2, d_2), \cdots, (v_r, d_r)$ does not violate c. The tuple $(v_1, d_1), (v_2, d_2), \cdots, (v_r, d_r)$ is a *support* for (v, d)

Enforcing *GAC* on a CSP problem P is performed by removing any value of the domain of a variable that is not supported in a constraint of the problem until a fixed point is reached, or an empty domain is found. In the latter case the problem P has no solution. It turns out however that enforcing GAC on (non-binary) CSP's is an intractable problem (unless *P=NP*). Indeed for a CSP with domain size bounded by d the best complexity that can be achieved for an algorithm enforcing GAC is $O(erd^r)$ where r is the largest arity for a constraint and e is the number of constraints in the problem [107].

Since it is computationally hard to enforce GAC (especially maintain GAC during search) in practise weaker propagation methods are usually defined for non-binary CSP. A propagation method M_1 is strictly stronger than another propagation method M_2 (M_2 is weaker than M_1) iff M_1 prunes as many values as M_2 in all CSP problems and there exists at least one CSP problem in which M_1 prunes more values than M_2. For example *Bound Consistency* (BC) is a weaker propagation method than GAC. *Bound Consistency* 'bounds' the domain of a variable (provided that the domain is a well-defined ordered set) by removing the maximum and minimum values from a domain if they are not supported. More formally (the definition is similar to *bound(D)* of [107] and as in [36]):

Definition 4 A CSP problem $P =< X, D, C >$ is Bound Consistent (BC) if and only if all its constraints c in C are Bound Consistent. A constraint c in C of P is Bound Consistent (BC) if the value $\min D_x$ and $\max D_x$ of each variable v in $scope(c)$ is consistent with the rest of the variables of $scope(c)$.

Enforcing *BC* on a CSP problem P is done by removing the minimum and maximum values of the domain of a variable that are not supported in a constraint of the problem until a fixed point is reached, or an empty domain is found. Although enforcing *BC* on general CSP problems is again intractable [107] it is usually much more efficient making a CSP problem *BC* than *GAC*.

The GAC defined in definition 3 for *binary constraint satisfaction problems* is often named as *Arc Consistency* (AC). It is extensively studied and used by most solvers for binary constraint satisfaction problems. Two efficient algorithms that enforce AC on binary CSPs are *AC6* and *AC2001* [107]. For a binary CSP with e constraints and domain size bounded by d the time complexity for both algorithms is $O(ed^2)$ and their space complexity is $O(ed)$ [107].

For *binary constraint satisfaction problems* more general forms of consistency have been introduced than *Arc Consistency*, such as the *Path Consistency* and *Inverse Path Consistency* [107], that belong to the general family of (i, j)-consistency. *Arc Consistency* also belongs to the family of (i, j)-consistency.

Definition 5 A CSP problem $P =< X, D, C >$ is *(i,j)-consistent* if for any consistent assignment to values for any set X_i, $X_i \subseteq X$ and $|X_i| = i$ there exists a consistent assignment to values for any set X_j, $X_j \subseteq X$,$|X_j| = j$ and $X_j \cap X_i = \emptyset$.

The time complexity of *(i,j)-consistency* is exponential in *i.j*. For $i = 1$ these methods are 'domain filtering' (they remove values from the domains of variables), but due to their high computational cost are rarely used in practice for large j. Enforcing *(i,j)-consistensy* needs exponential space in i since constraints of arity i must be added in the problem in order to forbid the inconsistent tuples. For this reason they are seldom used for small j and large i as well. *Arc Consistency* is the *(1,1)-consistency*. *Path Consistency* is the *(2,1)-consistency* and *Inverse Path Consistency* is the *(1,2)-consistency*. Methods that attempt to reduce the computational burden by restricting the consistency checks to a subset of the variables, include the Restricted Path Consistency (RPC), Max-Restricted Path Consistency (maxRPC) and Neighborhood Inverse Consistency (NIC) [107]. Their definitions can be found in [107].

Definition 6 A CSP problem $P =< X, D, C >$ is *strong (i,j)-consistent* if it is *(x,y)-consistent* $\forall x, 1 \leq x \leq i$ and $\forall y, 1 \leq y \leq j$

It holds that a CSP problem $P =< X, D, C >$ is globally consistent (satisfiable) iff it is *strong* *(*$|X|$*,1)-consistent.*

A consistency propagation method that is strictly stronger than *Arc Consistency* and is used in practise but does not belong to the family of *(i,j)-consistency* is the *Singleton Arc Consistency* (SAC). In the definition below $P|_{x_i=v}$ is the problem that is derived from the problem P, $P =< X, D, C >$ when the value v,$v \in D_{x_i}$ is assigned to variable x_i, $x_i \in X$.

Definition 7 A CSP problem $P =< X, D, C >$ is *Singleton Arc Consistent* (SAC) if for all $x_i \in X$ and for all $v \in D_{x_i}$ the problem $P|_{x_i=v}$ is *Arc Consistent.*

Enforcing SAC to a problem is achieved by repeatedly assigning a value to a variable from its domain and examining if the resulting problem is AC, otherwise the value is removed from the domain of the variable. The iteration halts when a fixed point or an empty domain is found (for a problem without a solution). For a problem with n variables e constraints and a domain size bounded by d, Bessiere and Debruyne [14, 15] proved that the time complexity of SAC is $O(end^3)$. They proposed an (optimal) algorithm (SAC-Opt) of time complexity $O(end^3)$ and space complexity $O(end^2)$ [107]. Another algorithm for SAC, is the SAC-SDS (SDS is for *Sharing Data Structures*) is of time complexity $O(end^4)$ and space complexity $O(n^2d^2)$ [107]. A family of singleton propagation methods strictly stronger than SAC can be defined in the obvious way by replacing the AC in the definition of SAC with a strictly stronger propagation method. For example replacing AC with *Path Consistency* (PC) in the definition 7 yields the *Singleton Path Consistency* (SPC) propagation method. Obviously SPC is strictly stronger than SAC and computational harder since PC is strictly stronger and computational harder than AC.

Figure 1 (as it is presented in [107]) summarizes the result for the qualitative comparison for the pruning power for *Strong Path Consistency*, (Strong PC), *Restricted Path Consistency* (RPC), *Max-Restricted Path Consistency* (maxRPC), *Neighborhood Inverse Consistency* (NIC), *Singleton Arc Consistency* (SAC), *Path Inverse Consistency* (PIC) and *Arc Consistency* (AC). The formal proofs can be found in [34].

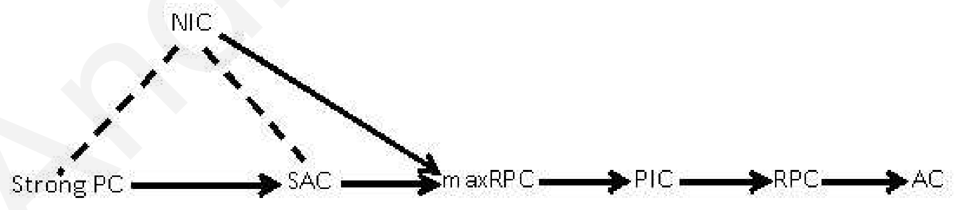

Figure 1: Summary of the comparison between consistecies AC and Strong PC. $A \rightarrow B$ means that the consistency A is stringly stronger than B and the dashed line means that are incomparable. The stronger relation is transitive.

Global Constraints [107, 36] are constraints that constrain all the variables of a CSP problem. They are semantically redundant in the sense that they can be expressed as the conjunction of simpler constraints. For example consider a CSP problem P containing n variables that are constrained to have pairwise different values. This obviously can be expressed by $\frac{n(n-1)}{2}$ binary constraints $x_i \neq x_j$, $\forall i, j \in [1 \ldots n]$ and $i \neq j$. This can be expressed equivalently by the global constraint $alldifferent(x_1, \ldots, x_n)$. Not only this is obviously better from a software engineering point of view, it is also better for constraint propagation. The reason is that there exist algorithms for enforcing *BC* [91] and *GAC* [107] on $alldifferent(x_1, \ldots, x_n)$ constraint that run in low polynomial time. Other global constraints that are widely used are the *GlobalCardinality-Constraint*, *sum* and *knapsack* constraints [107].

2.1.2 Backtracking Search Algorithms for deciding consistency

The most usual way to find a solution to a CSP problem P ($P =< X, D, C >$), especially if completeness is needed, is with a backtracking search algorithm. A backtracking search algorithm for solving a CSP problem [107, 36] can be seen as performing a depth-first traversal to a search tree.

Backtracking algorithms vary in the way backtrack is performed on dead-ends. In its simplest and earlier versions backtracking is performed chronologically, where the algorithm backtracks in the previous level in the search tree from the level of the dead-end [107, 36]. Newer versions of backtracking algorithms for CSP problems can backtrack (or backjump) deeper in the tree, using a conflict analysis mechanism, such as conflict-sets or no-goods [35], to preserve completeness.

Conflict-directed backjumping (CBJ) [107] in CSP solvers is achieved with no-goods or conflict-sets. The standard definition of no-good (or conflict set) was given in [35], as an assignment set that is not contained in any solution. Informally the conflict-directed backjumping works as follows: For each variable v of the problem is associated a conflict-set, say $conflictset[v]$ initially $\emptyset$. Each time a variable v' is responsible for eliminating a value from the domain of v due to assignment or propagation, $conflictset[v]$ is updated to $conflictset[v] \cup \{v'\}$. As the search proceeds, if for a variable say i no value can be assigned then the algorithm backjumps to the highest level over all variables of $conflictset[i]$, say j and it updates the $conflictset[j]$ to $(conflictset[j] \cup conflictset[i] \setminus \{j\})$. Completeness is reserved since if a dead-end is also found at j, the algorithm will backjump as high as the variable being in conflict with either i or j. The negation of the assigned values of variables in conflict can be stored as no-goods since they are inconsistent with any possible solution of the problem. This may help the propagation algorithm as the search proceeds. The backtracking strategy is combined with a constraint propagation algorithm to produce a family of CSP solvers. For example maintaining arc consistency on constraints with at least one uninstantiated variable in combination with conflict directed backjumping is the MAC-CBJ [107]. There are however two drawbacks in storing *no-goods*: Learning no-goods at each dead-end is too space-consuming and in may eventually slow down the propagation algorithm since it has to do a lot more consistency checks. Therefore two commonly used methods are storing relatively small no-goods, or periodically removing some of the no-goods based on a measure, such as the branches that passed since the last time they were used in propagation.

[67] implements a CSP algorithm that maintains consistency not in the separate constraints of the problem, but in conjunction of constraints that are grouped together using a heuristic measure. Obviously enforcing GAC in the conjunction of constraints may achieve more propagation than enforcing GAC on each constraint separately. Experiments showed improvement for some problems over the usual GAC. The idea of no-goods is further expanded in [68] to generalized no-goods. The main idea is that except of combining only assignments in a no-good and forbidding them from happening again, in a similar manner *not assignments* are considered as a no-good

(general no-good). A general no-good combines *not assignments* that are pruned by the constraint propagation along with normal assignments. General no-goods (g-nogoods) have the ability to store the same information as an exponential number of usual no-goods and they propagate much better. Experiments showed that g-nogoods in a CSP solver decreased significantly runtimes for crossword puzzles, planning and scheduling CSPs over the usual no-good scheme.

Another important feature of backtracking CSP algorithms is the heuristic that is used to pick the next variable and the value to branch on. The dom heuristic picks the next variable to instantiate having the less values in its domain. Other heuristics are $dom+deg$ and dom/deg. The heuristic $dom + deg$ [107] picks a variable following dom heuristic, but it breaks ties in favour of the variable maximizing deg, where deg is the number of constraints of the problem involving this variable and at least one other unassigned variable. The heuristic dom/deg [107] divides for each variable its dom value by the value deg and picks the variable minimizing this score. An extension of this heuristic is the $dom/wdeg$ [23] heuristic. Before search a weight associated with each constraint is initialized to one. As the search proceeds any time that a constraint becomes inconsistent its weight is incremented. The heuristic $dom/wdeg$ picks the variable that minimizes the division of the size of its domain (dom) divided by the sum of weights of constraints of the problem involving this variable and at least one other unassigned variable ($wdeg$). The $dom/wdeg$ seemed to perform very well in a variety of problems. The intuition behind all these heuristics is to pick a variable that is the most likely to fail first. Two of the most well known heuristics in the literature to select a value to assign to a variable are the 'promise' [107] heuristic and the 'min-conflicts' [107] heuristic. Given a variable x, the promise (min-conflicts) heuristic picks the value that after constraint propagation the product (sum) of the sizes of the domains of the remaining uninstantiated variables is maximized.

2.1.3 Solving Constraint Optimization Problems

One of the most common (complete) algorithms for CSP problems with objective functions is the branch and bound. For example in integer linear programming (ILP) the domains of all variables are subsets of $\mathbb{Z}$, the constraints are arithmetical linear inequalities and the objective function (to be minimized) is the scalar product of a weight vector of real or integer numbers with the variables of the problem P ($P =< X, D, C >$). The algorithm initializes a global variable I that holds the best solution found so far to ∞ and each time it finds a solution, as in a usual CSP, using backtracking and constraint propagation (usually maintaining bound consistency for arithmetic domains), it updates I to $min(I, f(X))$, where $f(X)$ is the value of the objective function for the current solution. The value of I is used during search to prune the domains of the variables and to backtrack. For a partial assignment on say k $(k < |X|)$ variables $X_1, X_2, \cdots, X_k$ of X, before assigning to an unassigned variable X_{k+1} a value v from its domain, it is checked that $f(X') < I$ (assuming that all domains are positive integers as well as the weights else the check would be a little more complex), where X' is the partial assignment on the variables $X_1, X_2, \cdots, X_k, X_{k+1}$. If the inequality does not hold, then the domain $D_{X_{k+1}}$ of variable X_{k+1} can be soundly updated to $D_{X_{k+1}} = D_{X_{k+1}} \setminus \{v\}$.

In CSP (and other hard) problems it is common to use relaxations. A relaxation is performed by relaxing the constraints of the problem P to obtain a more easy problem P' that can be solved efficiently. The gain (in CSP context) is that it can guide the search in the original problem e.g. used as a heuristic, it may filter some domains of unassigned variables or it may prune entire subtrees reducing therefore the search space. It can also be used for approximations. A common example is the integer linear programming problem (ILP) with an objective function to minimize. While being an *NP-complete* problem, if the integrality constraints are relaxed and each variable v_i can take a real number value in $[min(D_{v_i}) \cdots max(D_{v_i})]$ then the problem is a Linear Programming (LP) problem which can be solved very efficiently in polynomial time (e.g. [83]).

For example using a branch and bound algorithm to solve an ILP (minimization) problem, a relaxation of the corresponding *linear programming* (LP) problem is solved on each assignment of a variable (for the unassigned variables) which gives a value to the objective function f^*. f^* is the minimum value that can be achieved in the ILP problem as well when instantiating the rest of the variables, since $\mathbb{Z} \subset \mathbb{R}$. Assuming the value of the best found (integer) solution to be I, if $I \leq f^*$ the entire subtree can be safely pruned since no better solution can be found when instantiating the rest of the variables to an integer value. Moreover the solution of the dual LP problem of the relaxation can be used for domain filtering , by propagating inequalities [59].

2.2 Satisfiability of propositional formulas

SAT-solving is used to solve many hard problems of the real world. It is used in a variety of applications [18] such as bounded model checking for hardware [17] and software verification [32, 63], combinatorial designs [126], Satisfiability Modulo Theories [5, 48], product configuration [115] and planning.

The propositional satisfiability problem (or SAT) is a special case of a constraint satisfaction problem, where all the variables have as domain the $\{True, False\}$, and all the constraints are propositional formulas. Propositional formulas are propositions connected with the logical connectives $\neg(not), \vee(or), \wedge(and)$ and $\longrightarrow$ $(implies)$. A propositional formula is in *Conjunctive Normal Form* (CNF) if is a conjunction of clauses, where a clause is a disjunction of literals. A literal is a propositional variable or its negation.

2.2.1 Resolution and restricted forms of resolution

The simplest and perhaps the oldest algorithm for inference in propositional logic is resolution [59]. Given a formula $F = \{c_1, c_2, \cdots, c_n\}$ in CNF form resolution iteratively picks two clauses c_i and c_j from F that have a common literal, say p, in one clause c_i being positive and in clause c_j being negative, and adds the resolvent clause $c_r = (c_i \setminus \{p\}) \bigcup (c_j \setminus \{\neg p\})$ in the SAT theory until a fixed point is reached or the empty clause is derived. Resolution will return an empty clause if and only if the SAT theory is unsatisfiable. *Unit resolution* and *Binary resolution* are special cases of resolution. *Unit resolution* resolves two clauses if at least one of the resolvent clauses is a unit clause (contains only one literal) and *binary resolution* resolves two clauses if at least one of the resolvent clauses is a binary clause (contains two literals). *Unit Propagation* (UP) is the most common propagation algorithm in SAT solvers. It repeatedly applies *Unit resolution* in the input SAT theory until closure. The *Fail Literal Detection Rule* (FL-rule) [43] assigns the value true (false) to an unvalued literal l of a theory T if $UP(T \cup \{\neg l\})$ ($UP(T \cup \{l\})$) derives the empty clause. *Fail Literal Propagation* (FL-prop) is the propagation method that repeatedly applies the FL-rule until closure. Two restricted forms of binary resolution are *BinRes* [6] and *Krom-subsumption resolution* (KromS) [119]. *BinRes* resolves two clauses only if the resolvent clauses are both binary. *Krom-subsumption* resolution resolves a binary clause $c_i = xy$ with a clause of the form $c_j = \neg xyC$ (where x, y are literals and C is a clause) and adds the resolvent clause $c_r = yC$ which subsumes the clause c_j. The propagation method *BinRes-prop* (*KromS-prop*) applies the BinRes (*Krom-subsumption*) resolution rule in a SAT theory until a fixed point is found or the empty clause is derived.

2.2.2 SAT and CSP

There are a number of works for translating a problem from CSP formulation to SAT and vice versa, for example [123, 13] and [37]. Dimopoulos and Stergiou present in [37] *two* standard widely studied ways of translating a binary CSP to SAT the *Direct encoding* and *Suuport encoding* and *three* standard ways of translating a SAT to CSP the *Literal encoding*, *Dual encoding* and *Non-Binary encoding*.

To translate a (binary) CSP to SAT using the *direct encoding* or the *support encoding* for each value a for each variable x_i of the CSP is introduced a propositional variable x_{ia}. For each variable $x_i \in X$ with a domain $D(x_i), |D(x_i)| = d$ there is an at-least-one clause $x_{i1} \vee \ldots \vee x_{id}$ to ensure that x_i takes a value. They are also (optional) at-most-one clauses to ensure that each variable x_i takes at most once value from its domain: For each i, a, b such that $a \neq b$ and $\{a, b\} \subseteq D(x_i)$ is added a clause $\neg x_{ia} \vee \neg x_{ib}$. The difference between the *direct encoding* and *support encoding* lies in the clauses that are added to express the constraints of the CSP: For each binary constraint C on variables $\{x_i, x_j\}$ and for each a, b such that the tuple $< (x_i, a), (x_j, b) >$ is not allowed in *direct encoding* a binary clause $\neg x_{ia} \vee \neg x_{jb}$ is added, whereas in *support encoding* the constraints are captured by clauses that express the supports that values have in the constraints. For all values $a \in D(x_i)$ the clause $x_{jb_1} \vee \ldots \vee x_{jb_s} \vee \neg x_{ia}$ is added where $x_{jb_1}, \ldots, x_{jb_s}$ are the propositional variables corresponding to the s supporting values of a in $D(x_i)$ The *support encoding* is a better translation than *direct encoding* in respect to the propagation achieved. For example enforcing AC (SAC)in the CSP can achieve more propagation than UP-prop (FL-prop) in *direct encoding* whereas AC (SAC) are equivalent to UP-prop (FL-prop) in *support encoding* in respect to the propagation achieved.

To translate a SAT problem T to (binary) CSP using the *Literal encoding* for each clause $c_i \in T$ a variable v_i is introduced, and $D(v_i)$ consists of those literals satisfying c_i. For each pair of clauses c_i, c_j in T such that exists a literal $l \in c_i$ and $\neg l \in c_j$ a binary constraint is posted in the CSP for the variables v_i (that corresponds to the clause c_i) and v_j (that corresponds to the clause c_j). This constraint forbids the incompatible assignments for the two variables (e.g. (v_i, l) and $(v_j, \neg l)$). The translation of T to (binary) CSP under the *Dual encoding* turns out to be stronger than *Literal encoding* with respect to the propagation achieved. For example there is a direct correspondence between AC (SAC) and UP-prop (FL-prop) in *literal encoding* but AC (SAC) achieve more propagation than UP-prop (FL-prop) in the *Dual encoding*.

2.2.3 State of the art SAT solvers: non-chronological backjumping and learning

One of the first practical algorithms for the SAT problem is the *Davis-Putnam* algorithm [108], although the version implemented in most solvers is the 2-years later version called *Davis-Putnam, Logemann and Loveland* (*DPLL*). It is a chronological backtracking algorithm , branching on all unassigned variables recursively to *true* and *false*, and it performs *Unit Propagation* (UP) on each assignment. *DPLL* either finds that the SAT problem is unsatisfiable after building the whole tree, or finds a total satisfiable assignment of the variables.

There are more than two dozen of SAT solvers entering the SAT competitions every year that are based on DPLL along with some new important features such as the two watch literals scheme, conflict-learning and backjumping. Details can be found in [18] , [86] and [84].

The main algorithm for SAT solving that is used in most modern solvers is called 'conflict driven clause learning' (CDCL). The basic difference from DPLL is that instead of branching on a variable p and its negation, CDCL branches only on the one side (not necessarily always the same), (or assumes p=false), and if a contradiction is found then a literal that resolves this contradiction is forced (it may be p=true, but not necessarily), and a ’conflict clause’ or ’learnt-clause’ is learned (a no-good) and stored. The clause is the ’reason’ for the conflict, and may be used later for propagation.The most usual constraint propagation method used in SAT solvers is the Unit propagation (UP). A notable exception is the SAT solver 2CLS+EQ [7] which implements a stronger propagation method similar to Fail Literal propagation.

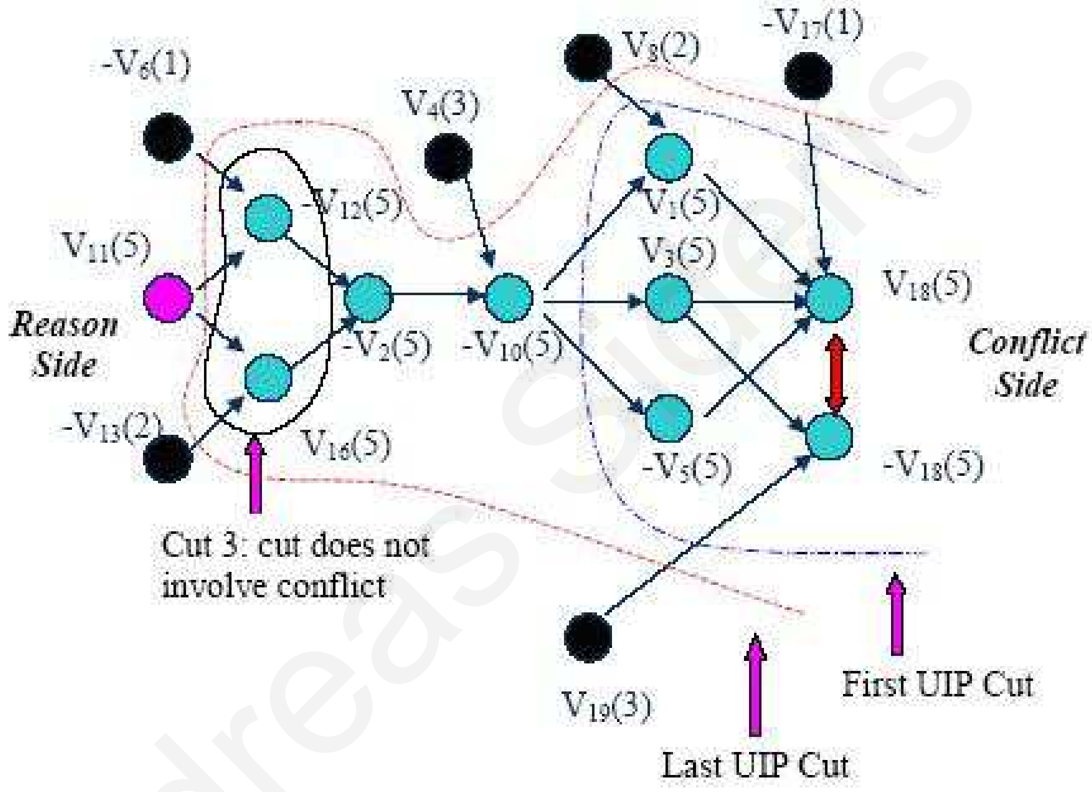

Figure 2: Implication Graph (figure from [127])

The most significant clause learning mechanisms of CDCL-based solvers use the implication graph, which is better illustrated graphically. Figure 2 is a typical implication graph (as presented in [127]). An implication graph [127] is a directed acyclic graph (DAG), with each vertex representing a variable assignment, a positive (negative) variable means it is assigned to true (false) (e.g. V_{16} is assigned to true and V_5 to false). The incident edges in the graph of each vertex correspond to the reasons that lead to an assignment. In the parenthesis next to each variable is the decision level at which was assigned its value either because it was the decision of that level, or because it was implied by UP at that level. The incident edges in the graph of each vertex correspond to the reasons that lead to an assignment. The implication graph is used when a conflict is found after the decision is unit propagated. Since a conflict is found this implies that a variable is assigned by UP to be true and false (variable V_{18} in the figure). Obviously the only interesting vertices in the implication graph are the ones which are connected through a path to one of the conflicting variables, the rest of the vertices (and edges) are irrelevant. In an implication graph a vertex a dominates vertex β iff any path from the decision variable of the current decision level (the pink variable V_{11} in the figure) to β must go through a. A Unique Implication Point (UIP) is a vertex of the current decision level that dominates both conflicting variables, for example V_{11}, V_2, V_{10} in the figure. A UIP is a single reason for the conflict. UIPs are sorted from the conflicting variables to the decision variable (which by definition is always a UIP, the last-UIP) as first-UIP, second-UIP, $\cdots$, last-UIP.

A conflict clause is derived by a bipartition in the implication graph, having all the decision variables on one side (the reason side) and the conflicting variables on the other side (the conflict side). All vertices of the reason side with at least one edge in the cut comprise the reason of the conflict and deduce a conflict clause (a no-good). Each cut corresponds to a valid conflict clause. However in order for the conflict clause to be an asserting clause (i.e. there is a unique literal of the current level (the blue vertices in the figure and the pink)), a partition needs to have exactly one UIP (it holds that one always exists - the decision variable) on the reason side, and all vertices assigned after this on the other side. The first-UIP cut and the last-UIP cut are showed in figure 2. The respective clauses are $\{V_{10}, \neg V_8, V_{17}, \neg V_{19}\}$ and $\{\neg V_{11}, V_6, V_{13}, \neg V_4, \neg V_8, V_{17}, \neg V_{19}\}$ with the asserted literals V_{10} and $\neg V_{11}$ respectively. In [127] different learning schemes were experimentally evaluated and first-UIP showed to be the best - perhaps because the asserting literal is as much close to conflict as possible. Chaff, Siege and MINISAT all use the first-UIP.

Different heuristics for deciding the next variable to branch on are used in different solvers. Some of these heuristics are MOMS, Jeroslow-Wang, VSIDS and VMTF. MOMS heuristic chooses the literal with Maximum number of Occurrences in Minimum Size clauses. Jeroslow-Wang heuristic chooses the variable with the maximal score $JW(l)$, where $JW(l)$ is calculated as: $JW(l) = \sum\limits_{l \in c} 2^{-|c|}$, where l, c stand for literal and clause respectively.

VSIDS (Variable State Independent Decaying Sum) is probably the first heuristic suited for CDCL solvers introduced in solver chaff [87]. It increases the score of each variable in a new conflict clause, dividing scores periodically. In this way variables relevant to recent conflicts are preferred. This idea is usually used in heuristics of CDCL solvers. MINISAT's [40] heuristic is similar to VSIDS, but no distinction is made between positive and negative variables because MINISAT branches only on negative values . VMTF (Variable Move To Front) is the heuristic used by the siege solver [109]. Despite being very easy to implement and very cheap to compute it is experimentally proved to be very efficient, perhaps one of the best among CDCL solvers. A score for each literal is maintained, initially set to the occurrences of the literals in formula. A list W with the variables of the input formula is created before search and is sorted such as v_0 is before v_1 if $score(v_0) + score(\neg v_0) > score(v_1) + score(\neg v_1)$. Each time a clause c is learned, the scores of literals in c are increased by one and eight of the variables (corresponding to literals) in c are moved in W so they are put in the front in arbitrary order (if $|c| < 8$ then all $|c|$ variables are moved). When a decision is made the nearest to the front variable v in W that is not singleton is picked, and is set to true (false) if $score(v) > (<) score(\neg v)$, ties being broken randomly.

CDCL SAT solvers may record a big number of conflict clauses until they terminate, causing memory problems or even slowing down the propagation, hence they usually have a mechanism to delete periodically some of the learnt clauses. Another feature of most of the CDCL SAT-solvers is the restart mechanism. After a criterion is satisfied before the algorithm terminates, the algorithm throws away the partial assignment (except from level 0) and starts from the beginning, usually maintaining (some) of the learnt clauses. For example siege restarts every 16000 conflicts.

2.2.4 Tractable subclasses and decomposition

Two tractable subclasses of the SAT problem are the 2-SAT and the renamable Horn clauses. 2-SAT is a formula for which it holds that the maximum arity of a clause is 2. 2-SAT is solved in polynomial time with the use of the implication graph of the formula [88]. The Horn clauses are clauses that contain at most one positive literal. Proving satisfiability for a formula containing only Horn clauses is a polynomial time problem [38]. Renamable Horn clauses is a set of clauses that can be transformed to Horn clauses, by replacing zero or more variables with their negation. A SAT formula can be decided if it is a renamable Horn (and if it is to be renamed) in polynomial time (e.g. [59]), hence satisfiability for Renamable Horn clauses is still polynomial.

[4] presents algorithms for reasoning with partitions for propositional and first-order-logic (FOL). Two kind of algorithms are presented to query an already partitioned logical theory (both for propositional and FOL) based on message-passing over the separators of partitions, one forward passing and one backward (query driven) passing. Extended theoretical results and proofs are presented proving a time complexity exponential in the size of larger partition and a space complexity exponential in the size of largest separator over the partitions. An algorithm (called LINEAR-PART-SAT) for a partition SAT formula is presented again with a time complexity exponential in the number of variables of the largest partition, and a space complexity exponential in the product of largest path in the partition tree multiplied with the maximum separator (common variables number) over all connected partitions. Finally the paper presents a greedy polynomial

algorithm (Split-Thy) that separates a SAT theory in partitions (as those described above assumed by LINEAR-PART-SAT).

[78] presents two algorithms combining Hypergraph decomposition with a SAT solver in order to solve real-world SAT instances more efficiently.The hypergraph's vertices are the clauses of the problem, and the hyperedges are variables connecting the clauses (hypervertices) that contains them (positive or negative). A separator created in this work for the hypergraph H is a set of hyperedges (hence a separator corresponds to set of variables) that when removed from the hypergraph create k unconnected partitions $H_1, H_2, \cdots, H_k$, and for each one of them it holds that it contains 15%-85% of the nodes of the initial hypergraph. Two methods are tested: creating a separator before search (ESD) and creating separators dynamically during search (DSD). ESD proved experimentally to perform quite well in runtimes, whereas DSD did not - although fewer decisions were made by the solver it did not pay off the overhead of separating at each step.

2.3 Pseudoboolean Optimization

A pseudo Boolean (*PB*) formula is a special case of CSP. A PB formula is a set of Pseudo Boolean (*PB*) constraints, where a PB constraint is an inequality of the form $\sum a_i x_i \geq b$, where x_i is a boolean variable (a $\{0, 1\}$ variable) and a_i, b integers. The Pseudoboolean Optimization problem is the optimization problem of maximizing (or minimizing) a linear objective function on (some) of the variables of a PB formula subject to the constraints of the formula being true.

Since *PB* optimization problems are a special case of *Integer Linear Programming* they are usually solved by *Operational Research (OR)* methods such as *Integer Linear Programming (ILP)* [125, 124] (*ILP*) solvers. Three *PB* optimization solvers that utilise logical reasoning methods are the Pueblo solver [110], `bsolo` [85] and MiniSat+ [41].

The Pseudo Boolean solver *bsolo* [85] integrates lower bound estimation procedures (assuming minimization) to discover bounds on the variables of the optimization function in order to prune the search space, adds cutting planes (Knapsack constraints) and performs non-chronological backtracking when a conflict is found. Two lower bound estimation procedures were tested: 1) the Linear Programming Relaxation (LPR) and 2) Lagrangian Relaxation (LGR). The Linear Programming Relaxation was experimentally verified to perform by far better than LGR or in case none of the two methods is used. Moreover LPR is used in the heuristic of the *bsolo* solver: Branching is restricted to variables with non-integer values in the LP solution. The variable picked is the one with the value closest to 0.5. In case more than one variable have value 0.5 in the LP solution, then the VSIDS heuristic of Chaff [87] is used.

Pueblo [110] is a hybrid method for solving Pseudo Boolean (*PB*) satisfiability and optimization problems. A pseudo Boolean constraint is expressed as a linear 0-1 inequality with positive integer coefficients. Pueblo combines cutting-plane analysis (and addition) to the formula from ILP with clause learning of SAT when finding an inconsistency to learn a no-good and decide the backtrack level. Experiments on benchmarks demonstrated better results than with ILP and SAT solvers alone (when the *PB* is translated to a SAT instance), showing the effect of combining the two methods.

MiniSat+ [41] is a Pseudo Boolean (*PB*) satisfiability solver that uses the SAT solver MiniSat [40] to solve the PB problem. It first ignores the optimization PB function $f(.)$, translates the PB inequalities to an equivalent CNF propositional formula and invokes MiniSat to solve the SAT problem. If the propositional formula is satisfiable and the objective function for the found assignment is $f(.) = k$, it then adds the inequality $f(.) < k$ (assuming minimization) to PB formula, translates to CNF and invokes the SAT solver again. This is done until the MiniSat finds an unsatisfiable formula, and therefore the solution is the last found with the last score. Our work for PSP and PSP−H planning systems presented in chapters 5 and 6 respectively use the translation module of MiniSat+ to translate the PB formula to CNF.

There are three ways that translate the PB-constraints to equivalent CNF clauses implemented in MiniSat+ [41], using BDD trees, adders and sorters. We briefly describe the first method of BDD trees, since it is the method we use in this thesis and we refer to [41] for the description of the other two methods. This method first translates the PB-constraint into a BDD tree, and then converts the BDD into clauses with the introduction of auxiliary variables. It is proved that the translation with BDDs maintains arc-consistency but in the worst case yields translations of exponential size. However when the PB-constraints are restricted to cardinality constraints, as in the *PSP* planning system, the size becomes polynomial: more precisely a cardinality constraint $x_1 + x_1 + \ldots + x_n \geq k$ results in a BDD tree with $(n - k + 1) * k$ nodes. The translation of the PB-constraint to a BDD tree is done by a straightforward procedure using dynamic programming which is very efficient for few variables. After the BDD tree is built it is treated as a circuit of ITE (*if-then-else gates*). As a circuit representation [41] it uses the reduced boolean circuits (RBC) [1]. The BDD tree is translated to clauses using the Tseitin transformation [118], which is a linear transformation of propositional formulas to CNF clauses. For the purposes of minisat+ a propositional formula is considered as a single-output, tree-shaped circuit. The idea of the transformation is to introduce a new auxiliary variable for each output of each gate. For example for the $OR(a, b)$ it introduces a new variable x for the output and CNF clauses are added corresponding to the relation $x \leftrightarrow (a \vee b)$.

2.4 Summary

In this chapter were briefly presented the most significant elements of constraint satisfaction, propositional satisfiability and pseudoboolean optimization. The two most important elements are constraint propagation and heuristics. Heuristic is the method of the solver that picks the next variable/value to branch on, based on an easily computable estimation of the ’goodness’ of each potential choice. A constraint propagation (CP) algorithm infers new constraints from the current partial assignment of the problem. A CP method is stronger from another if it infers more constraints. In the context of this work we invented a new way to translate the planning problems into propositional satisfiability formulas, the `SMP` presented in chapter 4. We formally prove that Unit Propagation, the constraint propagation mechanism in the vast majority of modern SAT solvers, achieves more propagation when `SMP` translation is used than other translations in the literature. This theoretical result is of practical use too, since better constraint propagation yields better runtimes as well, as our experiments on many different benchmark planning problems revealed. In chapters 5 and 6 we considered the planning problem as a Pseudoboolean optimization problem where the objective function is the maximization of the goals that are attained, subject to the constraints of the `SMP` translation. Experimental results revealed the practical use of the approach.

Chapter 3

Propositional Planning

Propositional (or classical) planning is planning in simplest form. As we noted in chapter 1, in propositional planning the world is finite and deterministic, actions have no durations or costs and are always executed as outlined by the plan. Moreover the state of the problem is changed only by the actions of the plan in a deterministic way and the initial state of the problem is fully known.

The description of a *propositional planning problem* consists of two parts: The *domain* and the *problem*. The *domain* is the 'abstraction' that describes how the actions change the state of the world. The domain specifies a high level description of the preconditions and effects of each action, as well as the set of all propositional symbols that model the world. The *problem* specifies the initial state and the goals of the problem. A planner takes as input both the *domain* and the *problem*.

3.1 Representations and Solutions of propositional planning

There are a number of ways to represent a classical planning problem, for example Ghallab, Nau and Traverso in [47] define the *set-theoretic representation*, the *classical representation* and the *state-variable* representation:

1. In a *set-theoretic representation* each sate of the world is represented as a set of propositions. Each action is represented as a syntactic expression of propositions that specifies when the action is applicable and what is the effect of the action i.e. what proposition(s) will add or remove in the following state.

2. In a *classical representation* actions and states are similar to ones in set-theoretic representation. The difference is that first-order literals and logical connectives are used, instead of propositions as in the set-theoretic approach.

3. In a *state-variable representation* each state is represented by a tuple of n state (or multi-value) variables $\{x_1, \ldots, x_n\}$. Each state variable x_i is associated with a finite domain. At each state the state variable x_i takes exactly one variable from its domain, or the 'undefined' value. A partial function that maps this tuple to another tuple of values of the n state variables is used to represent each action.

Each of the above representations has the same expressive power and can be easily transformed to any of the other [47]. We give the formal definitions of the set-theoretic representation as close to [47], whereas we follow the formalisms in [12] for the state-variable representation. The classical representation is not used elsewhere in this thesis, therefore we do not give any definitions. Formal definitions for classical representation can be found in [47].

3.1.1 Set-Theoretic Representation

Definition 8 A *set-theoretic planning domain* on a finite set of propositional symbols L is a restricted state-transition system $\Sigma = (S, A, \gamma)$ such that:

- $S \subseteq 2^L$. i.e. each state $s \in S$ is a subset of L. A proposition p holds in state s iff $p \in s$.

- Each action $a \in A$ is a triple $a = \{pre(a), add(a), del(a)\}$ where $pre(a) \subseteq L$, $add(a) \subseteq L$ and $del(a) \subseteq L$. Moreover $add(a) \cap del(a) = \emptyset$. The sets $pre(a)$, $add(a)$ and $del(a)$ are called the *preconditions*, *add effects* and *delete effects* of action a respectively. The action a is *applicable* to a state s if it holds that $pre(a) \subseteq s$.

- The set of states S has the property that if $s \in S$ then for all the actions a that are applicable to s, $(s \setminus del(a)) \cup add(a) \in S$. Informally this means that any action applicable to a state will produce another state.

- The state transition is defined as:

$$\gamma(s,a) = \begin{cases} (s \setminus del(a)) \cup add(a) & \text{if } a \text{ is applicable to } s \\ undefined & \text{otherwise} \end{cases}$$

Definition 9 A *set-theoretic planning problem* is a triple $P = (\Sigma, s_0, g)$ where:

- s_0 is the *initial state* and $s_o \in S$.

- $g \subseteq L$ is a set of propositions called the *goal propositions*. For any state s that is a *goal state* it holds that $g \subseteq s$. The set of all goal states is denoted as S_g.

Definition 10 A *sequential plan* π is a sequence of actions $\pi = \langle a_1, \ldots, a_k \rangle$. The length of the the plan π is denoted as $|\pi|$ and $|\pi| = k$, where k is the number of actions in the plan . The state that is produced by applying the plan π on a state s is the state that is produced if all the actions of π are applied on s in the order specified in π (from first to last):

$$\gamma(s,\pi) = \begin{cases} s & \text{if } k = 0 \\ \gamma(\gamma(s,a_1), \langle a_2, \ldots, a_k \rangle) & \text{if } k > 0 \text{ and } a_1 \text{ is applicable to } s \\ undefined & \text{otherwise} \end{cases}$$

Definition 11 A *sequential (or serial or total order) plan* π is a *solution* (or *valid (sequential) plan*) for a planning problem $P = (\Sigma, s_0, g)$ if $\gamma(s_0, \pi) \in S_g$. The plan π is a *sequential optimal* plan if there exists no valid plan π' for P such that $|\pi'| < |\pi|$.

It is often the case that some actions of a sequential plan can be executed simultaneously without affecting the soundness of the plan. For example in a logistics domain the two actions of loading two different packages in two different trucks can be executed simultaneously. However an action that loads a package cannot be executed simultaneously with an action that unloads the same package in a valid plan because they *interfere*.

Definition 12 Two (different) actions A_1 and A_2 *interfere* whenever any of the sets $del(A_i) \cap add(A_j)$ and $del(A_i) \cap pre(A_j)$ is non-empty $\forall i, j \in \{1, 2\}$ and $i \neq j$.

The state transition for a set of actions A to a state s is defined as:

$$\gamma(s, A) = \begin{cases} (s \setminus (\bigcup\limits_{a \in A} del(a))) \cup (\bigcup\limits_{a \in A} add(a)) & \text{if } \forall a \in A \text{ is applicable to } s \\ undefined & \text{otherwise} \end{cases}$$

Definition 13 A *parallel plan* π is a sequence of *sets* of actions $\pi = \langle A_1, \ldots, A_k \rangle$. The (parallel) length of the the plan $|\pi| = k$, where k is the number of action sets that are executed in parallel. The state that is produced by applying the plan π on a state s is the state that is produced if all the action sets of π are applied on s in the order specified in π (from first to last):

$$\gamma(s, \pi) = \begin{cases} s & \text{if } k = 0 \\ \gamma(\gamma(s, A_1), \langle A_2, \ldots, A_k \rangle) & \text{if } k > 0 \text{ and } \forall a \in A_1 \text{ is applicable to } s \\ undefined & \text{otherwise} \end{cases}$$

Definition 14 A *parallel plan* π , $\pi = \langle A_1, \ldots, A_k \rangle$ is a *solution* (or *valid (parallel) plan*) for a planning problem $P = (\Sigma, s_0, g)$ if $\gamma(s_0, \pi) \in S_g$ and $\forall A_i \in \pi$, such that $|A_i| > 1$ it holds that

$\forall(a_x, a_y) \in A_i \times A_i$, $a_x \neq a_y$, a_x and a_y *do not* interfere. The plan π is an *optimal* parallel plan or *step optimal* if there exists no valid parallel plan π' for P such that $|\pi'| < |\pi|$.

It is useful to note that a step optimal parallel plan $\pi = \langle A_1, \ldots, A_k \rangle$ for a problem may have more actions ($\sum_{A_i \in \pi} |A_i|$) than another sub-optimal plan. This is not the case in sequential plans where actions are executed sequentially. Sometimes *parallel plans* are also named as *partial order plans* in the literature since the actions of the plan are partially ordered, and not totally ordered as in sequential plans.

The combination of the definitions of *set-theoretic planning domain* and *set-theoretic planning problem* in definitions 8 and 9 respectively fully define a planning problem instance. However this combination cannot be used as an input to a planner because it explicitly gives all the members of γ and S that are exponential in the size of the problem. Therefore the *statement* of the problem $P = (I, A, G)$ is used instead. I is the initial state (denoted as s_0 in definition 9), G is the set of goals (denoted as g in definition 9) and A is the set of actions. It is easy to see that the statement is linear with respect to the size of the problem. Any new states that are computed by the planner due to the execution of an action(s) are created 'on the fly'.

3.1.2 State-Variable Representation

The definition of the State-Variable representation or multi-valued state representation of the planning problem that is presented is the SAS^+ of [12, 64]. The definition also corresponds to the *statement* of the problem in State-Variable representation of [47]

Definition 15 A SAS^+ *problem* (or *instance*) is defined as a tuple $\Pi = \langle V, O, s_0, s_* \rangle$ with components defined as follows:

- $V = \{v_1, \ldots, v_n\}$ is a set of *state variables*. For each state variable $v_i \in V$ is associated a *finite discrete domain* D_{v_i}. The extended domain $D^+_{v_i}$ of v_i is defined as $D_{v_i} \cup \{u\}$, where u stands for the *undefined value*. The total state space and partial state space are implicitly defined as $S_V = D_{v_1} \times \ldots \times D_{v_n}$ and $S_V{}^+ = D^+_{v1} \times \ldots \times D^+_{vn}$. $s[v_i]$ denotes the value of variable v_i in state s.

- O is a set of *operators* Each operator $o \in O$ is a triple $\langle pre, post, prv\rangle$ where $pre, post, prv \in S_V{}^+$, denote the *pre-condition*, *post-condition* and *prevail-condition* respectively. Every operator $o = \langle pre, post, prv\rangle \in O$ is subject to the restrictions **R1** and **R2** where:

  (**R1**) $\forall v \in V$ if $pre[v] \neq u$ then $pre[v] \neq post[v] \neq u$ and

  (**R2**) $\forall v \in V, post[v] = u$ or $prv[v] = u$

- $s_0 \in S_V{}^+$ denotes the *initial state* of the problem.

- $s_g \in S_V{}^+$ denotes the *goal state* of the problem.

Restriction R1 ensures that a variable cannot become undefined if was made defined before by an operator. Restriction R2 ensures that the prevail-condition of an action never defines a variable which is affected by the operator. Prevail-condition of an operator must hold (as pre-conditions) in order that the action can be executed, but are not affected by the operator as the pre-conditions. For example an operator *load* for loading a cargo in a truck in a specific location has as pre-condition the location of the cargo since the new location of the cargo after the operator is executed will change to *in* the truck. In contrast the location of the truck is a prevail-condition since it is not affected by the execution of the action. For the SAS^+ action $o = \langle pre, post, prv\rangle \in O$ its *pre-condition*, *post-condition* and *prevail-condition* are denoted by $pre(o)$, $post(o)$ and $prv(o)$ respectively.

For a SAS^+ problem $\Pi = \langle V, O, s_0, s_* \rangle$ a variable $s \in V$ is *defined* in a state $s \in {S_V}^+$ if $s[v] \neq u$ (where u is the *undefined* value). The state s is subsumed (or satisfied) by state t, noted as $s \sqsubseteq t$, if $s[v] = u$ or $s[v] = t[v]$. This notion ($\sqsubseteq$) is extended to whole states:

Definition 16 For a SAS^+ problem $\Pi = \langle V, O, s_0, s_* \rangle$ for any two states $s, t \in {S_V}^+$,

$s \sqsubseteq t$ if $\forall v \in V$ it holds that $s[v] = u$ or $s[v] = t[v]$

Definition 17 For a variable $v \in V$ and two values $x, y \in D_v$ such that $x = u$ or $y = u$ the operation $x \sqcup y$ is defined as:

$$x \sqcup y = \begin{cases} x & \text{if } y = u \\ y & \text{if } x = u \end{cases}$$

The above operation is extended to states as:

Definition 18 For two states $s, t \in {S_V}^+$, $(s \sqcup t)[v] = s[v] \sqcup t[v]$,$\forall v \in V$.

Definition 19 For a set of operators O the set of all operator sequences over O is defined recursively as:

$Seqs(O) = \{\langle\rangle\} \cup \{\langle o \rangle; \omega | o \in O\}$ and $\omega \in Seqs(O)\}$

where ; is the concatenation operator. All members $P \in Seqs(O)$ are called plans.

Definition 20 For two given states $s, t \in {S_V}^+$, $s \oplus t$ denotes that *t updates s* and is defined such that $\forall v \in V$

$$(s \oplus t)[v] = \begin{cases} t[v] & \text{if } t[v] \neq u \\ s[v] & \text{otherwise} \end{cases}$$

The state that results from applying a sequence of actions to a state s is given by function γ that is recursively defined as:

$\gamma(s, \langle\rangle) = s,$

$$\gamma(s, (\omega; \langle o \rangle)) = \begin{cases} \gamma(s, (\omega)) \oplus \text{post(o)} & \text{if (pre(o)} \sqcup \text{prv(o))} \sqsubseteq \gamma(s, (\omega)) \\ \perp & \text{otherwise} \end{cases}$$

where $\perp \in {S_V}^+$ is the state such as $\perp [v] = u, \forall v \in V$.

Definition 21 The relation *Valid* is defined recursively as the ternary relation $Valid \subseteq Seqs(O) \times {S_V}^+ \times {S_V}^+$ for any sequence of actions $\langle o_1, \ldots, o_k \rangle \in Seqs(O)$ and any states $s, t \in {S_V}^+$, $Valid(\langle o_2, \ldots, o_k \rangle, s, t)$ if either of the following two conditions is true

1. $k = 0$ and $t \sqsubseteq s$ or

2. $k > 0$, pre(o_1) $\sqcup$ prv(o_1) $\sqsubseteq s$ and $Valid(\langle o_2, \ldots, o_k \rangle, (s \oplus \text{post}(o_1)), t)$.

A plan $\langle o_1, \ldots, o_k \rangle \subseteq Seqs(O)$ is a *solution (or valid sequential plan)* for the SAS^+ problem $\Pi = \langle V, O, s_0, s_* \rangle$ if $Valid(\langle o_1, \ldots, o_k \rangle, s_0, s_*)$.

An *action* in the SAS^+ formalism is an *instantiation* of an operator. Given an action a, *type(a)* denotes the operator that a instantiates.

Definition 22 A *partial-order (or parallel) plan* is defined as the tuple $\langle A, \prec \rangle$ where A is a set of actions (i.e. instantiations of operators) and $\prec$ is a strict partial order on the set A. The *partial-order plan* $\langle A, \prec \rangle$ is a *solution* of the SAS^+ planning instance Π if for each topological sort $\langle a_1, \ldots, a_n \rangle$ of $\langle A, \prec \rangle$, it holds that $\langle type(a_1), \ldots, type(a_n) \rangle$ is a solution of Π.

Two very important structures for the State-Variable variable representation are the *Domain Transition Graph (DTG)* and the *Causal Graph*. Their definitions follow [51].

Definition 23 Consider a planning problem in the SAS^{+} with a set variable V. The *domain transition graph* G_v (DTG) of the variable $v \in V$ is the digraph with vertex set D_v^{+} and contains an $arc(d, d')$ ($d' \neq u$ where u is the undefined value) iff there is an operator $\langle pre, post, prv \rangle$ where either $pre[v] = d$ or $pre[v] = u$, and $post[v] = d'$.

Definition 24 Let $\Pi = \langle V, O, s_0, S_* \rangle$ a planning problem in the SAS^{+}. Its *causal graph* is the digraph (V, A) containing an $arc(x, y)$ iff $x \neq y$ and there exists an operator $o \in O$ such that $post[y] \neq u$ and either $prv[x] \neq u$ or $post[x] \neq u$.

3.2 Problem hardness and efficient solutions

The classical planning problem is *PSPACE-complete* [26] in general, and *NP-complete* for a pre-determined finite bound plan length [26]. There are a number of works that study this theoretical complexity. Two kinds of such works are decomposing the planning problem as in factored planning, and identifying special cases of planning domains which are polynomial solvable. The two approaches are briefly presented in this subsection.

3.2.1 Factored Planning

Factored planning was first presented in [3] by Amir and Engelhardt, although similar ideas in planning are much older, based on hierarchical planning. The idea in factored planning is to separate the planning domain in different components (or subdomains or factors -hence the name) which loosely interact, organized in a tree-structure. Then planning is done in each factor separately and finally the plans of the factors are synthesized to a plan for the whole domain. Planning in a subdomain can be done by any planner. The challenge in factored planning is how to factor the domain in different factors, how to organize these factors and how to plan using these factors, especially how to synthesize the solution. Since factors are organized in a tree structure, it's easy to see that the worst time complexity becomes exponential in the size of the largest factor plus the size of interactions between the factors. Two specific algorithms that follow this approach are *PartPlan* [3] and *LID-GF*[25]. Both algorithms use tree decomposition, with the actions being distributed among all the factors. Planning is performed recursively, by computing all possible plans in all the subtrees of a factor, and then the subplans are merged, and planning proceeds in the current factor. This is performed from the leaves factors up to the root factor where the final plan (if any) will be returned. Although algorithms report good theoretical results, [25] especially reports worst case complexity being exponentially better from [3], there are no experiments on benchmarks for neither of the algorithms. Moreover minimum plan length is not guaranteed and excessive memory is needed for many planning domains.

[77] presents the factor planning algorithm *dTreePlan*, which is the first algorithm for factored planning with backtracking. *dTreePlan* is a complete optimal (in respect to planning horizon) planner and it uses a decomposition tree to factorize the planning domain, using a tool based on hypergraph partitioning. The nodes of the hypergraph are the actions of the problem and two nodes are in the same hyperdge if they share at least one common variable (e.g. a precondition). The hypergrpaph is recursively balanced bi-partitioned and planning is performed starting from the root factor, from a factor to children factors. Experiments using the planning as satisfiability approach (with SAT solver chaff) proved quite promising.

3.2.2 Syntactic and Structural restrictions

There are some special cases of propositional planning that are provably solvable in polynomial time. An approach to find such subclasses of propositional planning is enforcing a number of 'restrictions' on the planning domains and then devising dedicated deterministic polynomial planning algorithms for these classes of planning domains. There are two types of such restrictions, the syntactic restrictions and the structural restrictions. Algorithms for syntactic restricted problems assume the SAS^{+} representation of the problem and syntactic restrictions on the definition of the problem. Bäckström and Klein [11] described four syntactical restrictions *Post-unique (P)*, *Unary (U)* , *Binary (B)* and *Single-valued (S)*. A problem is P iff for each value for all state variables of the problem, there is at most one action having it as post-condition, and is U iff all actions change the value of only one state variable in their post-conditions. The problem is B iff all state variables have domain of size two and S if there do not exist two actions that have as a prevail condition different (both defined) values of the same state variable.

Bäckström and Klein [11] proved that optimal planning of $SAS^{+} - PUBS$ class can be performed in time $O(m^3)$, where $SAS^{+} - PUBS$ are problems under P,U,B,S restrictions and m is the number of state variables of the problem. Relaxing the restriction B, Bäckström presented a polynomial planning algorithm [10, 9] for $SAS^{+} - PUS$ optimal planning with time complexity $O(m^2 n)$, where n bounds the domain size of state variables. Bäckström and Nebel present a polynomial algorithm for $SAS^{+} - US$ (sub-optimal) planning in [12] with a time complexity $O(|M|^4|A|)$ where M is the set of state variables and A is the set of operators of the problem.

Planning under structural restrictions [64] again assumes the SAS^{+} formalism, but this time the restrictions are on the Domain Transition Graph (DTG) of each state variable. Five structural restrictions are defined with respect to the DTGs of the problem: *Interference-safeness (I)*, *Acyclicity (A)* with the variants A^{+} and A^{-} depending on the restrictions and *Prevail-order-preservation (O)*. A^{+} is more restrictive and A^{-} less restrictive than *Acyclicity (A)*. A SAS^{+} planning problem is restricted from any of the above structural restrictions if all the DTGs of the problem satisfy the restriction. The definitions of the five restrictions are found in [64]. A complete algorithm of time complexity in $O(|V|^2|O|^2M^5)$ is presented in [64] that tests if a SAS^{+} problem satisfies the IAO structural restrictions and then finds a plan (if the IAO restrictions are satisfied) in a time complexity $O(|V|^3|O|^2M^5)$, where V and O are the sets of state variables and (grounded) actions respectively and M is the largest domain size among all the state variables of the problem.

Brafman and Domshlak [24] present another tractable subclass of STRIPS planning. Assuming again the multivalued variables representation of the problem they present a non-trivial tractable planning algorithm for plan generation for this subclass of planning problems. For this class of problems the *Unary (U)* and *Binary (B)* syntactical restrictions hold, together with the restriction that the *causal graph* of the problem is a polytree (a causal graph is a polytree if the induced undirected graph is acyclic) with indegree of all nodes in the graph bounded by a constant k. The authors present a sound and complete algorithm (procedure FORWARD-CHECK) for this class of problems that returns false if a plan does not exist. Otherwise the algorithm first topological sorts the causal graph and then calculates the valid changes in values for all variables and the order they must be conducted from the root to the leafs in a plan. Then FORWARD-CHECK passes as input this order and the operators needed for the changes in a modified deterministic Partial Order Planner (POP-PCG) that finds a plan in linear time (without backtrack). The time complexity of FORWARD-CHECK is $O(|V|^{2k+3}2^{2k+2})$ for a problem with a state variable set V. Another important result proved in [24] is that plan generation for STRIPS planning problems under the *Unary (U)* syntactical restriction and acyclic causal graph is provably intractable (harder than *NP*).

3.3 Planning algorithms

3.3.1 The Graphplan planner

Graphplan [20] builds a structure called the planning graph of a planning problem and then searches for a subgraph that satisfies certain conditions in order to be a valid solution (valid plan of the problem).

3.3.1.1 The planning graph: Reachability and mutual exclusion relations

A planning graph [20, 47] of a classical (propositional) grounded planning problem is a compact representation of the state space of the problem. It is a directed layered graph with two kind of nodes and three kinds of edges. The layers (also called levels) of the graph alternate between proposition levels and action levels, containing proposition and action nodes respectively. The first level of the planning graph is a proposition level containing the propositions that are in the initial state of the problem. Then a layer with all applicable actions is added (these having all their preconditions in the previous layer), then a layer with propositions that are the add-effects of actions in previous layer, and so on. The levels from the earliest to the latest are propositions that are possibly true at time 1 and actions that are possibly executable at layer 1, propositions that are possibly true at layer 2 and actions possibly executable at layer 2 and so on. The three kind of edges are precondition edges, add-edges and delete-edges. Precondition edges connect actions of layer i with their preconditions in level i and add-edges (delete-edges) connect actions of level i with the propositions of level $i + 1$ in their add-list (delete-list). A special kind of action, a 'dummy' No-op(p) action is added for each proposition p to resolve the frame problem, having empty delete-list, and preconditions and as add-effects the set $\{p\}$.

One of the powerful features of the planning graph is the ability to find and propagate (some) mutually exclusive (mutex) relations between actions and propositions of the same level. Two facts are mutually exclusive in a layer if they cannot be simultaneously true in any valid plan, whereas two actions are mutually exclusive in a layer if they cannot be simultaneously executed in any valid plan. More formally:

Definition 25 Two actions A_1, A_2 of a planning problem $P = \{I, A, G\}$ are *mutually exclusive* (or *mutex*) at planning graph layer t if they *interfere* or $\exists p_1 \in pre(A_1)$ and $\exists p_2 \in pre(A_2)$ such that facts p_1, p_2 are *mutually exclusive* at layer $t-1$ for all valid layers of the planning graph.

Definition 26 Two propositions p_1, p_2 of a planning problem $P = \{I, A, G\}$ are *mutually exclusive* (or *mutex*) at layer t of the planning graph if any action A in the planning graph of layer $t-1$ such that $p_1 \in add(A_1)$ is *mutually exclusive* with all the actions $A_1, \ldots, A_n$ such that $p_2 \in add(A_i)$, $1 \leq i \leq n$ of layer $t-1$ for all valid layers of the planning graph.

Recall from definition 12 that two actions *interfere* if the one deletes a precondition or an add effect of the other. The mutexes are propagated in the planning graph from the initial state to the final layer of the planning graph. Starting from layer 1, the only possible mutex pairs are interference actions. At layer 2 two propositions are marked as a mutex pair if each action in layer 1 that has the one proposition as an add-effect is mutex with each one of the actions having the other proposition as an add-effect. The pairs of actions that are marked as mutexes at layer 2 are those that interfere, or a precondition of the one is marked as a mutex with a precondition of the other. The marking of the mutex pairs of facts and actions proceeds in this way from one graphplan layer to the next. The size of the planning graph is polynomial and is built in polynomial time in respect to the size of the problem [20]. A planning graph 'levels off' at a layer l, if l is the first layer for which it holds that all the facts and actions of layer l exist in layer $l-1$ and all the mutex relations for facts and actions of layer $l-1$ exist in layer l as well. All the layers of the planning graph that follow l are the same both for variables and edges. A planning graph for a small example problem is illustrated below.

Example 1 Assume a small problem domain containing one truck with a crane, two containers and two locations. The truck can move from one location to the other location, load a container if the container is in the same location as the truck and the truck is empty, and unload it to the location of the truck provided the container is loaded in the truck. The truck can hold only one container. At the initial state the truck T is in location L_2 whereas both containers are in location L_1. The goal is to transfer both containers C_1, C_2 from location L_1 to location L_2. More formally this problem P, $P =< I, A, G >$ is defined as follows:

$I = \{at\_C_1\_L_1, at\_C_2\_L_1, at\_T\_L_2\}$

$G = \{at\_C_1\_L_2, at\_C_2\_L_2\}$ and

$A = \{Move(T, L_1, L_2), Move(T, L_2, L_1),$

$Load(T, C_1, L_1), Load(T, C_2, L_1), Load(T, C_1, L_2), Load(T, C_2, L_2),$

$UnLoad(T, C_1, L_1), UnLoad(T, C_2, L_1), UnLoad(T, C_1, L_2), UnLoad(T, C_2, L_2),$

where the action descriptions are:

$pre(Move(T, L_i, L_j)) = del(Move(T, L_i, L_j)) = \{at\_T\_L_i\}, add(Move(T, L_i, L_j)) = \{at\_T\_L_j\}$ for all $i, j \in \{1, 2\}, i \neq j$.

$pre(Load(T, C_i, L_j)) = \{at\_C_i\_L_j, at\_T\_L_j\}, del(Load(T, C_i, L_j)) = \{at\_C_i\_L_j\}$

$add(Load(T, C_i, L_j)) = \{in\_C_i\_T\}$ for all $i, j \in \{1, 2\}$.

$pre(UnLoad(T, C_i, L_j)) = \{at\_C_i\_L_j, in\_C_i\_T\}, del(UnLoad(T, C_i, L_j)) = \{in\_C_i\_T\}$

$add(UnLoad(T, C_i, L_j)) = \{at\_C_i\_L_j\}$ for all $i, j \in \{1, 2\}$.

Figure 3 illustrates the planning graph for this example.

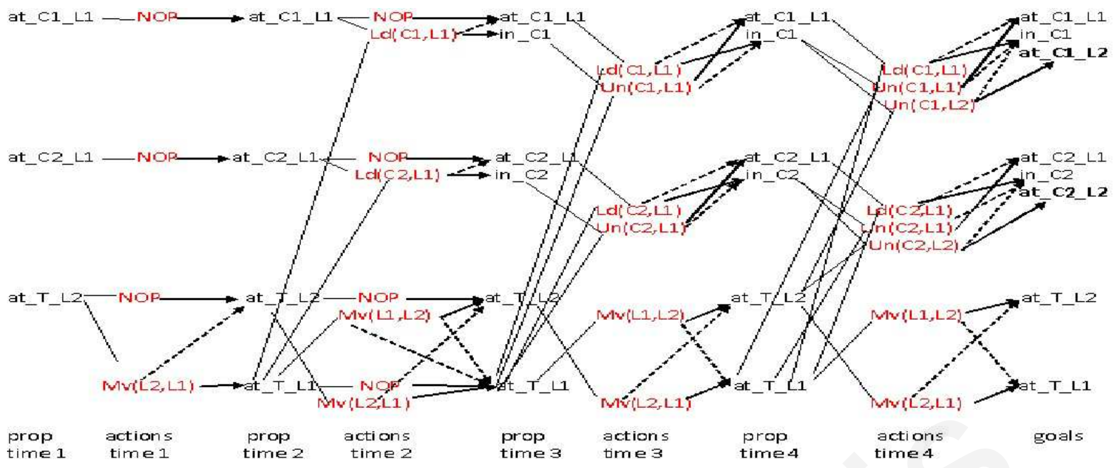

Figure 3: The planning graph for horizon 4 for the problem of example 1. For better readiness $in\_C_i\_T$, $Move(T, L_i, L_j)$,$Load(T, C_i, L_j)$ and $UnLoad(T, C_i, L_j)$ are illustrated as in_C_i, $Mv(L_i, L_j)$,$Ld(C_i, L_j)$ and $Un(C_i, L_j)$ respectively. Precondition edges are represented by lines and add (delete) edges by arrows (dashed arrows). Goals are noted in bold. No-op actions (presented as NOP at time steps 1 and 2) of time steps 3 and 4 are not presented for better readiness.

3.3.1.2 Extending the planning graph and searching for solutions

The Graphplan planner builds an initial planning graph of k levels, where k is the minimum number of levels (propositions and actions) such that in the last layer (after the actions of level $k - 1$) all goals of the problem are present and not mutex. Then the planner searches the planning graph using a backwards-chaining strategy to find a solution subgraph. It searches the graph level by level, that is, starting from the goals at time t, it searches all the actions (including No-ops) of time $t - 1$ having these goals as an add-effect. If such a set of actions is found (without any pair of them being mutex) supporting the goals, then their preconditions at time $t - 1$ become the new goals and the algorithm searches actions that support them at time $t - 2$ and so on, until a set of actions in time 1 is found. An action is not selected if it is mutex with all actions making a goal true ahead in the list. If no solution is found after searching the planning graph, the graph is extended with another level extending the graph with actions applicable to the previous goal layer, and then a layer with their postconditions being the new layer. Then the extended graphplan is searched for a solution. While searching the graph if a set of (sub)goals at a time point t is found to be unsolvable it is stored in a hash table indexed on t as a 'no-good' (a method called memoization) in order to be used later for pruning. These no-goods are also used for the termination condition of the graphplan planner, when a plan does not exist [20].

3.3.2 Graphplan and Constraint Satisfaction

In [65] the planning graph structure is considered as a dynamic CSP (DCSP) with variables being the propositions of the planning graph. A dynamic CSP is an extension of the CSP such that each variable of the problem is associated with an activity flag, indicating whether a variable is active or not. An implementation in LISP of CSP techniques was built over graphplan conceptualized as the described DCSP problem, including explanation-based learning, dynamic variable ordering, forward checking and non chronological backtracking, yielding to speedups over when those techniques were not used.

In [81] the planning graph is directly compiled to a CSP, but the action mutexes that are added correspond only to the actions that interfere, not to competing needs. It is shown (both theoretically and in practice) that by enforcing an initial (2, 2)-consistency the (binary) constraints added are a strict superset of the constraints that correspond to the action mutex constraints. Experimental results using a general CSP solver showed a significantly better performance in most benchmarks over Graphplan planner.

3.3.3 Graphplan and SAT: The BLACKBOX, SATPLAN and MaxPlan planning systems

An approach similar to the compilation of the planning graph to a CSP is the compilation of the planning graph to SAT. The first work in this context is due to Kautz & Selman in [72] back to 1992. In [73] the same authors achieved orders of magnitude speedups over Graphplan (which was one of the best optimal domain independent planners at that time) in many domains by using the local search SAT algorithm walksat. A combination of Graphplan and SATPLAN (SATPLAN is the framework for planning in SAT regardless of the SAT solver) was created in the Blackbox system [75].

There has been extensive research regarding what is considered to be a 'good' (automatic) translation of a planning domain to a SAT theory. Besides the obvious need for a correct translation regarding soundness and completeness, a translation is considered better than another if leads in an improvement of the solution times when employed by a modern SAT solver. Compilations are often characterised by a tradeoff between the number and size of clauses and the number of variables. Usually keeping the number of variables low increases the number or size of clauses and vice versa. After the innovative work in [72], the 'classical' paper is considered to be [71], where various translations of propositional planning domains to SAT are presented and theoretically analyzed. In [42] more encodings from those in [71] are described, as the result of an automatic translator implementation. In the community today more translations are based on translating the planning graph of the problem, as in Blackbox, SATPLAN and MaxPlan planners. There are some works explaining the planning graph construction and 'connecting' the planning graph with other encodings of the problem. In [44] Geffner proves that there is an exact correspondence between the computation of the planning graph and the iterative computation of prime implicates of size one and two over the logical encoding of the problem with the goals removed. In [99] Rintanen proves that the construction of the planning graph is covered by a restricted form of clause learning (of clauses of size 2) on the logical parallel encoding of the problem (again with the goals removed).

Blackbox [75] was the first system that combined Graphplan and SATPLAN. In Blackbox, Graphplan creates a planning graph, and then either a SAT solver or the Graphplan algorithm solves the problem. In fact, when the system was first released it used a variety of SAT solvers, such as the systematic solvers satz and rel_sat and the local search SAT solver walksat.

The SATPLAN framework is one of the best optimal domain-independent planners available today. The general idea is to first construct the planning graph, then translate this graph into a SAT theory, which is solved by a SAT system. There are differences in the compilations of the planning graph to SAT. For example SATPLAN in 2004 considered only action variables [72], whereas in 2006 competition SATPLAN used a different SAT compilation with action and fact variables. The new SMP encoding implemented in the context of this research, which is described in next chapter, speeds-up the planning procedure. Different encodings are presented in detail in chapter 4.

MaxPlan [128] is a another system based on SATPALN framework. MaxPlan achieves optimality in the opposite direction: It first finds a suboptimal plan invoking a fast suboptimal heuristic planner (the Fast Forward) and then it repeatedly reduces the makespan of the plan finding a plan for each one until it proves unsatisfiability. The search engine is a SAT solver that is 'customized' (based on *minisat* solver [40]) to perform better in planning domains under 2004 SAT encoding that it uses. Among other features it employs a different heuristic (preferring variables in non-binary unsatisfied clauses), and implied binary clauses (long distance mutual exclusion - londexes [30]). We study extensively londexes [30] in chapter 4.

3.3.4 Madagascar planner

The Madagascar planner [102, 101] is also based on the well known framework of planning as satisfiability [72]. The most significant feature of the planner is a novel heuristic used in the SAT solver, that exploits the underlying structure of the planning problem.

The variable selection mechanism is based on the simple property that all plans share: All problem goals must be made true by actions, as well as the preconditions of these actions must be made true by (other) actions unless they are true in the initial state. The first step in the selection of a decision variable is to find the earliest time point at which a goal (or subgoal) l for time t can become and remain *true*. This is done by going backwards from t to time point $t' < t$ for which one from the three cases below holds:

1. An action that makes true l is taken. In this case the plan has an action that makes l true.
2. Literal l is *false* and is either *true* or *unassigned* thereafter. In this case an action that makes l true in $t' + 1$ is chosen, and is used as the new decision variable.
3. The third case is that the initial state is reached and l is in the initial state, hence nothing is done in this case.

This simple heuristic was experimentally proved to speed up SAT planning for satisfying instances, but did not performed so well for unsatisfied instances against the VSIDS heuristic of the CDCL planner. The overhead to calculate the heuristic is not much greater than VSIDS. Refinements implemented in this heuristic include a (sub)goal ordering based on an estimation of which (sub)goal is most likely to be true first. The earliest a (sub)goal is estimated to be true, the higher the priority that is assigned. Moreover an action ordering is implemented. For an action a of a time point t the action ordering is based on the number of time points t' following t such that the variable a in t' is unassigned. The action with the minimal score is then selected. Rintanen also experimented with computation of several actions: that is, computing a set of some fixed number of actions and then picking randomly one of them.

The encoding that is used in Madagascar planner is the $\exists$-step encoding [103]. In traditional encodings (called $\forall$-step encodings in [103]), actions that interfere cannot be true in the same parallel step and therefore are constrained using binary constraints (mutexes) in the encoding. The $\exists$-step encoding relaxes the traditional encodings by requiring that there is at least one total ordering of the parallel actions and hence the encoding is linear in the number of action effects. The planner also uses powerful invariant algorithms as in [100] to discover any invariants (facts that are in initial state and will remain true after any number of actions) to further reduce the encoding.

The planning procedure used in the planner is not the usual solve and expand method [72] for increasing planning horizons, but instead is the algorithm B formalised in [97] with parameter $\gamma = \frac{9}{10}$. The algorithm interleaves the solution of several horizon lengths simultaneously and the satisfiability test for a formula of horizon n can be started (and also completed) before a test for a horizon less than n ends. This is important since the algorithm is not stuck in large unsatisfied formulas. Algorithm B allocates CPU time to formulae $\Phi_1, \Phi_2, \cdots, \Phi_n$ for different planning horizons proportional to γ^n. At any given time n is at most 20. If some instance of a horizon say k is proved unsatisfiable, the algorithm immediately discards any threads for horizons less than k and starts new threads. The planner plans for horizon lengths being integer products of 5, that is 0,5,10 and so on.

Experiments that were performed proved the effectiveness of the planner. The planner proved to lift the efficiency of SAT-based planning close to the same level with earlier best planners. In chapter 6 we experimentally compare Madagascar planner with our planner `PSP-H` which also uses SAT-solving to solve the planning subproblems that arise in the planning problems.

3.3.5 State Space Planners

The simpler and older planners are the state-pace planners. They are actually search algorithms that perform a search in the state-space of the problem. Many modern state space planners (some of which will be presented in this section), enhanced with powerful heuristics, are among the best available planners.

3.3.5.1 Forward and Backward Search

There are two 'traditional' ways to search for a plan in the state space, forward search and backward search. Given a planning problem with the descriptions of operators and a planning domain with an initial state S_0 and a goal set G, forward search begins from S_0 and moves forward trying to reach a goal state. It begins with the empty plan; if $G \subseteq S_0$, then the goals of the problem are satisfied and no plan is needed, else it expands S_0 via an action(s) to a state S' (this is a branching point). A plan is found if a state is reached being a superset of all problem goals. Backwards search searches for a plan in the opposite direction, from the goals to the initial state.

3.3.5.2 Heuristics for Forward state-space planning: Relaxation Methods

Most of the heuristics developed for forward search-state planners are based on the rational intuition to proceed forward 'as close to goals as possible'. Since classical planning is PSPACE-complete more domain-independent heuristics are derived from approximations, or relaxations, of the planning problem. The most common and useful domain-independent relaxation [45] is the *delete-relaxation*. The *delete-relaxation* maps a propositional planning problem $P = (I, A, G)$ to the problem $P^+ = (I, A', G)$ where the set of actions A' is the same as the set of actions A but $\forall \alpha' \in A'$ the $del(\alpha')$ is set to $\emptyset$. Although optimal planning for P^+ is *NP-hard*, sub-optimal planning is polynomial [26]. Two heuristics based on the idea of *delete-relaxation* are the *additive* and the *max* heuristics [45]. For a problem $P = (I, A, G)$ the *additive* heuristic of a state of the problem s is defined as:

$$h_{add}(s) = \sum_{g \in G} h_{add}(g; s)$$

where:

$$h_{add}(p;s)) = \begin{cases} 0 & \text{if } p \in s \\ \min\limits_{a \in Add(p)} (1 + h_{add}(Pre(a);s)) & \text{otherwise} \end{cases}$$

and

$$h_{add}(Pre(a);s) = \sum_{p \in Pre(a)} h_{add}(p;s)$$

In the expressions above $h_{add}(p;s)$ is the estimated cost of achieving fact p from s, $Pre(a)$ represents the preconditions of action a and $Add(p)$ the actions having p in their add effects. The *max* heuristic $h_{max}(s)$ is defined exactly as above with the difference that the $\sum$ operator is replaced with the operator $\max$. Although the h_{max} is admissible and h_{add} is not, variations of h_{add} are used in many planners such as in [22, 116]. The reason is that h_{max} is less informative than h_{add} and therefore is seldom used in practice. Another heuristic that is extremely simple and works surprisingly well for some domains as the *visitall* is the *number of unachieved goal heuristic* ($h_{ug}(s)$) that simply counts the number of unachieved problem goals in the state s. Many state-planners exploit the graphplan structure to calculate heuristics, such as the HSP planning system [21] and Fast Forward (FF) [55, 57]. Any of these heuristics can be used to evaluate states in a forward search planner built as a *heuristic search algorithm* , such as *Best-First Search*, A^* and *Hill Climbing* [45].

Katz et al. [70] proved tractable fragments of STRIPS planning when the delete lists are set to $\emptyset$ for *some* (instead of *all* as in *delete-relaxation*) of the actions of the problem. In the *red-black* relaxation of the problem (in the state-variable representation) $\Pi = \langle V, O, s_0, s_* \rangle$ the set of variables V is divided to *two* disjoint sets, the 'red' variables V^R that take the relaxed semantics and the 'black' variables V^B that take the regular semantics. Any action a affecting a state variable v updates $s[v]$ to $s[v] = (s[v] \cup post(a)[v])$ if $v \in V^R$ and to $s[v] = \{post(a)[v]\}$

if $v \in V^B$. Katz et al. in [70] proved *two* tractable fragments under the red-black relaxation. The first one requires a fixed number of black state variables and fixed domain size for each one of them. The planning algorithm for this fragment is exponential in $\prod_{v \in V^B} |D_v|$ which is usually too large to be used in practice. The second result they proved in the same work is that *plan existence* for red black relaxation of a problem Π is tractable if Π is *reversible* and the size of the largest strongly connected component of its *black causal graph* (the subgraph of causal graph over the V^B variables) is bounded by a constant. The red black relaxation problem Π is *reversible* if for any reachable state s from s_0, $\forall v \in V^B$ there exists an action sequence ω such that $\gamma(s, (\omega))[v] = s_0[v]$. This result also cannot be used in practice because tractability is proved for *plan existence* and testing *reversibility* for Π is *co-NP-hard*. In a later work [69] the authors relaxed the (above) definition of *reversibility* of a red black planning problem to the weaker definition of *RSE-invertible*. This (weaker) definition is based on the existence for any action a affecting a black variable of an 'inverse' action a' . The red black problem is *RSE-invertible* if all $v \in V^B$ are *RSE-invertible*, and a problem can be verified to be *RSE-invertible* in polynomial time. The authors proved that *plan generation* for a red black *RSE-invertible* planning problem with *acyclic black causal graph* is polynomial. They implemented an algorithm based on the above result inside the FastDownward framework (a state space planner) that for each candidate successor state s builds (and polynomially solves) a red black relaxation for the problem in order to calculate the heuristic score of s. Experiments on the domains from the IPC showed better runtimes and reduced search spaces of the red-black relaxation than the delete relaxation.

3.3.5.3 Landmarks

Informally a landmark is a fact that must be true at a time point in any valid plan of a problem. Obviously initial conditions and goals are trivially landmarks. If f is found to be a landmark, then if f being true implies that f' must be true at some point before f in any valid plan, then f' is also a landmark and the order relation $f' < f$ holds. An action landmark is an action that is present in any valid plan of a problem.

Hoffmann, Porteous and Sebastia first introduced landmarks in [90, 58]. Because finding all landmarks is PSPACE-complete [90], approximations are usually used. [90, 58] presents a two step backchaining process that identifies (some) of the landmarks and (some) of the orders between them. The landmarks and their orders are stored in a directed acyclic graph, the *landmarks generation graph* (LGG). In a first step the LGG graph is computed from the goals backwards to the initial facts. The goals are the first candidate landmarks. Then for any candidate landmark L', any fact L'' that has not yet been processed and is in the intersection of the preconditions of the 'earliest' (closest to initial state) actions that have L' as an add effect is added as a candidate landmark with the edge $L'' \rightarrow L'$ in the LGG graph. After the first step, in a second step any candidate landmark L (and its incident edges) is removed from the graph if the relaxed planning problem without actions having L as an add effect and ignoring all the delete lists from rest of the actions is unsolvable. After computing the LGG graph, in [58] the landmark graph was used in conjunction with FastForward (FF) and LPG [46] planners (that were state of the art at the time) to speed up planning. First the LGG graph is computed, and then the LGG graph passes the landmarks as disjunctive goals to the base planner (LPG or FF) which runs as an independent procedure. All the leafs of the LGG graph (nodes without incoming edges) are passed as a disjunctive goal to the base planner, and are removed from the LGG (and their incident edges). The algorithm iterates until LGG becomes empty, and then the original goals of the problem are passed to the base planner. Experimental results on various domains showed improved performance in runtimes against the base planners without using the landmarks graph, but often with longer plans.

Zhu and Givan in [129] use the planning graph to find *causal* landmarks of a planning problem in polynomial time. A fact landmark is causal if any solution plan contains an action having this fact as a precondition. In order to find causal landmarks, a planning graph is build, but more information (instead of only mutex fact/actions pairs), called labels, that are sets of action landmarks and facts landmarks are propagated during the extension of the planning graph. They used landmark extraction with some enhancements that are described in the paper in a heuristic planner they called LC. LC achieved slightly worse times when FastForward planner (FF) (being the state of the art at that time) performed well, but was much faster when FF performed badly.

In [95] Richter, Helmert and Westphal use landmarks to derive a pseudo-heuristic and combine it with other heuristics in a search framework with remarkable results in planning time, as well as in planing quality (shorter plans). Their approach is based on the SAS^+ formalism [12] of the planing problem, and they find landmarks in a similar way as in [89] but different in a few ways. The algorithm is adapted to SAS^+, and they use a different approximation for first achievers as well as a more general approach to find disjunctive landmarks (with maximum set size 4) as described in [89]. Moreover, by using the $DTG's$ of the SAS^+ formalism, they discover more landmarks. If for a variable v every path in the corresponding DTG from the value of v in the initial state towards a landmark value d' passes through a value d then d is also a landmark and d is ordered before d', noted as $d < d'$. Finally the order $A < B$ holds for any two landmarks A and B already found if B is not found to be *possibly before* A [89] (where *possibly before* is an approximated sufficient condition). The use of the landmarks found in the pseudo-heuristic is presented in section 3.3.5.6.

Karpas and Domshlak in [66] present the usefulness of landmarks in a best-first search algorithm for cost-optimal planning (actions in cost-optimal planning are associated with non-negative costs). The authors used the method described in [95] to approximate landmarks and their ordering, in conjunction with action landmarks in [129]. Their system, the *LM-A** planning system, was experimentally found to be very effective for cost-optimal planning.

3.3.5.4 FastForward planner

The *FastForward* planner (*FF*) [55, 57] is a forward search state space planner. The heuristic of FF [55, 57], $h_{FF}(s)$ evaluates the state s by building a graphplan with an initial state s and counts the number of all actions (not the No-ops) to reach all the goals of the problem. The FF planner starts from the initial state and uses *enforce hill climbing* to find a state with better evaluation than the current state (starting from initial state). On a state s, it evaluates all the direct successors, where a direct successor of a state s is a state that is produced by the execution of an applicable action. In the case that none has a better heuristic score from current, it looks at all the two-step successors and so on until a better state s is found, and the path of actions from s to s' is added to the plan. If FF gets stuck in a local minima it switches to weighted A^*[108] from scratch (replans from the initial state). Two other techniques of the *FF* planner that are used to prune the search space in the enforced hill climbing are the *helpful actions* and *added goal deletions*. The set of *helpful actions* $H(s)$ of a state s is defined as $H(s) := \{\alpha | pre(\alpha) \subseteq s, add(\alpha) \cap G_1(s) \neq \emptyset\}$ where $G_1(s)$ is the set of goals that is constructed by the relaxed graphplan at time step 1 with initial state (at time step 0) the state s. During enforced hill climbing, if the current state is s, the only successors that are considered are those that emerge from actions of the set $H(s)$. The *added goal deletions* technique eliminates any state s from the expansion list if in the relax solution plan from s there is an action that deletes a goal (in action's not relax version). The *FastForward* planner was experimentally proved to be among the fastest planners of its time for many different domains.

3.3.5.5 FastDownward planner

The FastDownward planner (FD) [52, 53] shares some similarities with the FF planning system. FastDownward like FF planner is a state space forward search (or progression) planner. However in contrast to other state space planners which use the propositional representation of the planning, FD translates the input (the translation procedure is described in [39]) in a sate-variable representation. FastDownward prior to search exploits this representation by using hierarchical decompositions of planning tasks in order to build the *causal graph* to compute a heuristic function, the *causal graph heuristic*. In order to compute the *causal graph heuristic*, first a relaxed acyclic graph is computed from the actual causal graph. Then the computation of the causal graph heuristic is performed by traversing the graph in a top down direction starting from the goal variables without causal predecessors. The algorithm for calculating the heuristic is based on the Dijkstra's algorithm for the single-source shortest path problem. Since the algorithm for calculating the heuristic is an approximation, it can be the case that the heuristic score of a non-dead end state is found to be (wrongly) ∞. This happens rarely in practice therefore *FD* treats these states as dead ends. However if *all* states have score ∞ FD implements a sound method to verify that they are real dead ends. If they are, it reports that the problem is unsolvable, otherwise it swhiches to the FastForward heuristic. Details for the computation of the heuristic can be found in [51].

The heuristic score is used to perform search in three different search algorithms, *greedy best first search algorithm*, *multi-heuristic best-first search* and *focused iterative-broadening search*. The system also supports the use of *preferred operators* (helpful actions) as in FF system in an orthogonal manner with the *causal graph heuristic*. *Greedy best first search* is the standard algorithm [108] modified with a technique to mitigate the effects of wide branching. *Multi-heuristic best-first search* is a variation of greedy best first search that uses multiple heuristic estimators including the heuristic of FastForward, maintaining separate open lists for each one. *Focused iterative-broadening search* instead of using heuristic estimators focuses on a limited operator set derived from the causal graph in order to reduce the set of search possibilities.

3.3.5.6 LAMA planner

The LAMA planner [96, 94] is a highly optimised implementation of a forward heuristic search planner that builds on the FastDownward planning system. LAMA is a state of the art in propositional planning. Its core feature is the use of a pseudo-heuristic based on the landmarks, in combination with a variant of the FastForward heuristic [55, 57]. LAMA (as the FastDownward system) consists of three separate components: **I)** The *translation* module that transforms the input PDDL problem into state variables representation, **II)** the *knowledge compilation* module that builds the *Domain Transition Graphs* and other structures that will be used in search, and **III)** the search module that performs the actual planning. The two first modules are used in other planners as well, such as the *PSP-H* planning system implemented in the context of this thesis and presented in chapter 6.

The method for finding ladmarks in LAMA planner is briefly described in section 3.3.5.3 as it appears in [95]. LAMA uses landmarks in heuristic search to estimate the goal distance from a state s. This number is estimated as $\widehat{l} = n - m + k$ where n is the total number of landmarks, m is the number of landmarks that are *accepted*, and k is the number of accepted landmarks that are *required again*. A landmark is *accepted* in a state (and will remain *accepted* in all successor states) if it is true in that state and all landmarks ordered before it are accepted in the predecessor state. An accepted landmark is required again in state s if it is not true in s and is the greedy-necessary predecessor of a landmark which is not accepted. This heuristic was used in the FastDownward framework [52] in combination with a variant of h_{FF} heuristic and *preferred* *operators* (an operator is *preferred* if it achieves an acceptable landmark) in an orthogonal manner. Experimental results in a variety of domains verified the improvement both in planning time as well as plan quality (shorter plans) against not using landmark information but also against older ways of using landmarks, as in [58].

There are two algorithms for heuristic search implemented in LAMA planner:

1. A greedy best-first search that finds a solution as quickly as possible.
2. A *weighted* A^* search which allows a balance between speed and solution quality.

The greedy-best first search always expands the state with minimal heuristic score among all candidate states, breaking ties in favour of states that are reached by cheaper operators. Details for the planner as well as the extensive experimental results showing the effectiveness of the planner are found in [96].

3.3.5.7 PROBE planner

The PROBE planner [80, 79] is a standard, complete forward search Greedy Best First Search (GBFS) planner that employs the usual additive heuristic to expand a new state. The key difference from GBFS is that before expanding a new state it launches a 'probe' from the current state to reach the problem goals. A probe is a sound polynomial procedure that starting from a state attempts to reach the goals through a sequence of actions (plan). If the probe reaches the goals the PROBE planner returns the plan, whereas in the case that the probe fails, each state visited by the probe procedure is added to the open list of the algorithm and a state is selected using the additive heuristic.

The probe procedure first approximates the landmarks and their order from the starting state towards the goals in a very similar manner as the LAMA planner. Then it tries to find a sequence of actions (without search) from the next unachieved landmark to the following in the order. The selection of actions is done in a greedily way, trying to find the most 'suitable' action without violating any commitments. An implementation of the PROBE planner in C++ experimentally proved to be competitive with the state of the art planner LAMA. Moreover a single probe from the initial state solved 683 out of 980 problems of previous planning competitions (in 2011) without any search, which compares well with the 627 problems solved by the FF planner.

3.3.6 Partial Order Planners (POP). CPT and eCPT planners

Partial order planners (POP) plan in a different way from state-space planners. Their search space is not the state space of the problem. The search space of POP planners is the plan space. Initially a 'plan' which is invalid either because actions are missing or the order of actions is inconsistent etc. is created, and gradually the plan is refined by adding actions and reordering them in the plan until a valid plan is found, if one exists.

3.3.6.1 Causal links, threats, partial orders and partial order planning

A partial plan is a tuple [47] $\pi = (A, <, B, L)$, where:

- $A = \{a_1, a_2, \cdots, a_k\}$ is a set of (partially) instantiated planning operators.
- $<$ is a set of binary ordering constraints on variables of A.
- B is a set of binding constraints on the variables of A.
- L is a set of causal links of the form $a_i[p]a_j$ (followed the symbolism in [121] instead of [47]) such than $\{a_i, a_j\} \subseteq A$, $(a_i < a_j) \in <$, $p \in pre(a_j) \cap add(a_i)$ and the binding constraints for actions a_i and a_j appearing in p are in B.

The plan space is the space of partial-plans, and searching the plan-space corresponds to performing refinement operations. These refinement operations are **1)** add an action to A, **2)** add an ordering constraint in $<$, **3)** add a binding constraint in B and **4)** add a causal link in L. An action a_k *threatens* a causal link $a_i[p]a_j$ if $a_i < a_k$ and $a_k < a_j$ and a_k has a delete effect p or an add effect q such that p and q are inconsistent under the binding constraints. A *flaw* in a partial plan $\pi = (A, <, B, L)$ is either a subgoal i.e. a problem goal or the precondition of an action in the plan without a causal link, or a threat to a causal link. A plan $\pi = (A, <, B, L)$ is a solution-plan (a valid plan to the problem) iff **1)** the $<$ and B constraints are consistent, and **2)** it has no flaws. A partial order planner (POP), or Partial Order Causal Link (POCL) planner may be implemented as a backtracking algorithm. The POP maintains a list of the flaws of the problem that is trying to resolve, which are open goals needed to be supported (a fact is supported if an action in A adds it) and threats. Initially the flaws list contains the goals of the problem. At each recursive step a flaw is selected, and the algorithm branches to all possible ways of resolving the flaw, e.g. adding causal links, ordering or binding constraints. The preconditions of any new actions added become open goals and are added to the flaws list with any new threats. A valid plan is found if the flaws list becomes $\emptyset$.

3.3.6.2 Heuristics for POP planners

POP planners choose a flaw that must be resolved, and a resolver to resolve that flaw. Obviously, this corresponds to a search in an AND/OR tree with alternating levels of flaws and resolvers where AND branches are the flaws (since all must be resolved) and OR branches are the resolvers (since only one is needed to resolve a flaw) [47]. Hence a heuristic for selecting a flaw to be resolved is the Fewest Alternatives First (FAF) heuristic that is selecting the flaw with least resolvers (as being the most constraint one) [47]. A simple heuristic for choosing a resolver is selecting the revolver that minimizes $|g_\pi|$ when applied, where g_π is the set of propositions in the partial plan π without causal links [47]. A more informative heuristic, that is based on a relaxation calculated by building the planning graph of the problem before search, is described in [47].

3.3.6.3 Constraint propagation for optimal partial order temporal planning

Vidal and Geffner implemented two Optimal Temporal Partial Order Causal Link (POCL) planners, the *CPT* [121] and the more extended *eCPT* [120] systems. The main differences/contributions in their approach over the standard POP planning framework (as was briefly described) is the enhancement with new powerful pruning mechanisms, being able to propagate constraints and prune domains for actions that are not part of the active partial plan. These propagation methods are powerful enough so that many of the classical planning benchmarks in the 2nd and 3rd International Planning Competitions were solved backtrack-free, with run times competitive to other state of the art solvers of that time as BLACKOX and SATPLAN for propositional planning domains (all action durations are set to 1 in the *CPT* and *eCPT* to 'simulate' classical planning). Although other components of the planners such as branching and heuristics are interesting and significant, they are not presented here and details are found in [120, 121].

eCPT planning system is a temporal POP planning system built over the POCL framework. Each action is realized as a Strip operator with add-list, delete-list and precondition-list and with a constant integer duration. The basic formulation of eCPT can be described in *four* parts, *preprocessing*, *variables*, *constraints* and *branching*.

At *preprocessing* phase the heuristic values $h_T^2(\alpha)$ and $h_T^2(\{p, q\})$ for each action and each pair of atoms are computed as in [50], providing lower bounds for the starting time of α and the starting time of the pair $\{p, q\}$ (the calculation of the family of tractable heuristics h^m can be found in [49]). h_T^1 is used to calculate minimum distances between actions: For two actions α

and α', a lower bound approximation for the starting times of α and α' denoted as $dist(\alpha, \alpha')$, is calculated and stored.

Variables and their domains are of four kinds:

1. $T(\alpha) :: [0, \infty]$ encodes the starting time of action α (being in $[0 \cdots \infty]$)
2. $S(p, \alpha)$ encodes the action supporting the precondition p of action α with domain all actions of the planning problem having p as an add effect
3. $T(p, \alpha) :: [0, \infty]$ encodes the starting time of $S(p, \alpha)$
4. $InPlan(\alpha) :: [0, 1]$ indicating if α is in the plan ($T(Start)$ is set to zero, and $InPlan(Start)$ as well as $InPlan(End)$ are set to true at initialization)

Constraints basically correspond to disjunction rules and precedences or their combination, to temporal constraints being propagated by bounds consistency. Constraints are of five kinds:

1. Bounds: stating that for each action α

   $T(Start) + \delta(Start, \alpha) \leq T(\alpha)$ *and* $T(\alpha) + \delta(\alpha, End) \leq T(End)$. (where $\delta(\alpha', \alpha)$ is defined as $dur(\alpha') + dist(\alpha', \alpha)$ and $dur(\alpha)$ is the duration of action α in the problem specification)

2. Preconditions: stating that

   $T(\alpha) \geq \min_{\alpha' \in D[S(p,\alpha)]} (T(\alpha') + \delta(\alpha', \alpha))$ (where $D(X)$ denotes the domain of variable X).

   $T(\alpha) \geq T(p, \alpha) + \min_{\alpha' \in D[S(p,\alpha)]} \delta(\alpha', \alpha)$

   $T(\alpha') + \delta(\alpha', \alpha) > T(\alpha) \rightarrow S(p, \alpha) \neq \alpha'$

3. Causal link constraints: stating for all actions α and p being a precondition of α that is e-deleted by action α', then

$$(T(\alpha') + dur(\alpha') + \min_{\alpha'' \in D[S(p,\alpha)]} dist(\alpha', \alpha'') \leq T(p, \alpha)) \bigvee (T(\alpha) + \delta(\alpha, \alpha') \leq T(\alpha'))$$

4. Mutex constraints: stating that for effect interfering actions [120, 121] α and α':

   $$(T(\alpha) + \delta(\alpha, \alpha') \leq T(\alpha')) \bigvee (T(\alpha') + \delta(\alpha', \alpha) \leq T(\alpha))$$

5. Support constraints: relating $T(p, \alpha)$ and $S(p, \alpha)$ by

   $$S(p, \alpha) = \alpha' \rightarrow T(p, \alpha) = T(\alpha')$$

   $$T(p, \alpha) \neq T(\alpha') \rightarrow S(p, \alpha) \neq \alpha'$$

   $$\min_{\alpha' \in D[S(p,\alpha)]} T(\alpha') \leq T(p, \alpha) \leq \max_{\alpha' \in D[S(p,\alpha)]} T(\alpha')$$

Except for the above constraints, a number of other constraint propagators is used in eCPT. These are *impossible supports*, *unique supports*, *distance boosting*, *qualitative precedences* and *action landmarks*. *Impossible supports* may eliminate at preprocessing time actions from the domains of $S(p, \alpha)$ variables. For example if action α' may support the precondition p of action α which has another precondition p' being e-deleted by α' (hence must be re-established), and all other actions supporting p' are incompatible with causal link $\alpha'[p]\alpha$, then α' cannot be a support of α and $S(p, \alpha)$ is updated to $S(p, \alpha) \setminus \{\alpha'\}$. In *unique supports* binary constraints of the form $S(p, \alpha) \neq S(p, \alpha')$ are posted for actions α and α' both having p as a precondition and as a delete effect. The lower bounds on the distances of actions α, α' computed at preprocessing (indicated as $dist(\alpha, \alpha')$) can be 'boosted' (hence the name *distance boosting*). For example although the distance of action *putdown(b_1)* and *pickup(b_1)* (b_1 being a block in a block domain) is $dist(\alpha, \alpha') = 0$, it can be safely updated to $dist(\alpha, \alpha') = 1$, since action *pickup($b_1$)* cancels *putdown($b_1$)* (an action α cancels action α' if every fact added by α' is e-deleted by α and every fact added by α is a precondition of α'). eCPT planner along with temporal precedences incorporates *qualitative precedences* for actions α, α' of the form $\alpha < \alpha'$ for all actions, even those not in the plan. The transitive closure of the qualitative constraints is being kept as an invariant in the planner and is used to further propagate constraints with the use of the following *two* inference rules : **1)** for an action α' in the plan adding a precondition p of action α if $\alpha \prec \alpha'$ then $S(p, \alpha) \neq \alpha'$ and **2)** for an action α' adding a precondition p of action α and an action b in the plan that e-deletes p: if $\alpha' \prec b$ and $b \prec \alpha$ then $S(p, \alpha) \neq \alpha'$. eCPT planner includes *action landmarks* at the preprocessing step, that are approximated by building a relaxed planning graph for each action. An action α is a landmark if removing it from the domain makes a goal unreachable. If when removing action b the action landmark α becomes unreachable then b is also an action landmark and moreover they are ordered as $b \prec \alpha$.

3.4 Summary

In this chapter the most significant elements of propositional planning were briefly presented. We presented some necessary preliminary definitions and properties of propositional planning problems and their solutions, but we emphasised on the presentation of propositional planners and the key elements that these planners share. Such planners are the Graphplan planner, planners that translate the planning graph in CSP or SAT, state space (or heuristic) planners and partial order planners. The work in this thesis is related to planning as satisfiability framework, that is building a planning graph, translate it to SAT and invoke a SAT solver in order to find a plan for successively larger planning horizons until a plan is found. More precisely, the `SMP` planner presented in chapter 4 follows the general planning as satisfiability framework but translates the problem to SAT in a novel more efficient way (wrt constraint propagation and runtimes of SAT solving). The new ideas for planning as pseudoboolean optimization in chapters 5 and 6 also rely on the `SMP` translation and SAT solving. Heuristics in state space planners usually rely on relaxations, for example the *delete relaxation*. The `PSP-H` planner implemented in the context of this work presented in chapter 6 is an attempt to incorporate ideas, such as the relaxation, from heuristic search planners into SAT-based planners (as is `PSP-H`).

Chapter 4

Constraint Propagation in Propositional Planning. The **SMP** system

In this chapter we present a comparison between different encodings of planning as satisfiability wrt the constraint propagation they achieve in a modern SAT solver. This analysis explains some of the differences observed in the performance of different encodings, and leads to some interesting conclusions. For instance, the BLACKBOX encoding achieves more propagation than the one of SATPLAN06, and therefore is a stronger formulation of planning as satisfiability. Moreover, our investigation suggests a new more compact and stronger model for the problem, called SAT-MAX-PLAN (abbreviated as SMP). The SMP encoding is a strict superset of the strongest among the three BLACKBOX encodings, which is the BB-31. SMP dominates all the encodings of SATPLAN06 and BLACKBOX wrt unit propagation, as we formally prove in this chapter. We also prove that in SMP many of the londex constraints are redundant in the sense that they do not add anything to the constraint propagation achieved by the model. Experimental results suggest that the theoretical results obtained are practically relevant. Finally we investigate experimentally the effects of adding more implied non-redundant binary constraints to the SAT encoding (in SMP), and we found strong evidence suggesting that it does not bring substantial gains wrt run times.

4.1 Introduction

As discussed in Chapter 3, after many years of research, there exist nowadays many different encodings of propositional planning as satisfiability, including those of `BLACKBOX` [75] and `SATPLAN06` [76]. In most of the studies these formulations are compared experimentally, and little is known about their theoretical underpinnings and the reasons that render one model better than the other. This thesis presents a first *theoretical analysis* that compares some of these encodings and explains important reasons that contribute to the differences which are observed in their performance.

Our investigation is based on the simple observation that the planning as satisfiability framework regards the planning problem as a Constraint Satisfaction one. Therefore, *constraint propagation* is a central notion. Stronger forms of propagation derive more variable values, and therefore lead to more pruning of the search space than weaker ones. If the computational cost of the constraint propagation procedure is low, the reduction of the search space usually translates into better run times. On the other hand, the addition of constraints that do not contribute to constraint propagation very often degrade the performance of systems, as they slow down the propagation of constraints as well as other techniques that may be employed in a modern SAT solver such as the clause learning.

This work compares different planning encodings wrt the *unit propagation* they achieve, the standard constraint propagation method employed in almost all modern SAT solvers. Roughly speaking, a planning model is stronger than another if it is able to propagate more variable values. Moreover, one encoding is more compact than some other if it achieves the same propagation but with a strict subset of the clauses. The clauses that are contained in the less compact encoding are *redundant* wrt unit propagation.

Our analysis reveals some interesting relationships. The most unexpected is probably that the BLACKBOX encoding is stronger than the one used in SATPLAN06. Based on our theoretical results we propose a new encoding of planning as satisfiability, called SAT-MAX-PLAN (abbreviated as SMP), that achieves more propagation than all other models, and it does so with a set of clauses that contains no redundancy.

We also study the propagation power of *long distance mutual exclusion constraints* (londex), as introduced in the MAXPLAN system [30], and show that they indeed strengthen the model of the SATPLAN06 encoding. More precisely, we prove that SATPLAN06 can propagate londex type information forward through the layers of the propositional theory, i.e. from variables that refer to a time point to variables that refer to some later time point. However, SATPLAN06 fails to do the same backwards, and therefore adding londex type constraints to SATPLAN06 encodings improves propagation. However, we show that londex constraints do not increase the propagation of the SMP encoding, and are therefore redundant in this new model. This is not the case for the type of londexes introduced in [29] that are stronger than those introduced in [30]. We prove that are not redundant in SMP encoding, and hence are not redundant in BLACKBOX and SATPLAN06 encoding as well.

In the experimental part we compare SMP, BLACKBOX, and SATPLAN06 in a number of domains from planning competitions. It turns out that SMP outperforms both other encodings, whereas between the two, BLACKBOX has an advantage over SATPLAN06. We also show that SMP compares favorably to the more recent SASE planner [60, 61, 62]. In fact, SMP coupled with the SAT solver precosat [16], can solve more problems than the all other planners, and presents a notable advancement of the state-of-the-art of planning as satisfiability. Moreover, it shows that the theoretical results of this work are of practical relevance. We also provide strong experimental evidence that adding more implied not redundant (wrt to Unit Propagation in SMP) binary constraints does not yield better run times.

To the best of our knowledge the only other studies close to the spirit of this work are [44] and [99]. However, the focus there is on understanding mutexes and londexes, and explaining how they can be derived by a modern SAT solver. Our investigation is complementary to the above, and explains *what*, *when* and *why* constraints improve performance.

4.2 Satisfiablity Encodings of Planning

In this section we discuss some of the most successful models of planning as satisfiability for optimal parallel planning. They include different encodings supported by the planning systems BLACKBOX, SATPLAN06, and SATPLAN04. We also discuss the more recent works of *split action model* of [106, 104] and the *SASE encoding* [60, 61, 62] based on the multi-valued representation of variables. We assume STRIPS planning problems $P =< I, G, A >$ as presented in Chapter 3. Recall that each action $a \in A$ has preconditions $pre(a)$, add effects $add(a)$, and delete effects $del(a)$.

All these systems use information that is derived from the planning graph of the problem. Part of the information that is extracted has the form of mutually exclusive pairs, or mutexes, as defined in section 3.3.1.1. The satisfiability encodings can be divided in three categories according to the semantics of the variables that are used in the SAT model of a planning problem.

1. *Direct encoding*. In a planning graph, each level corresponds to a different time point, while inertia is captured by *noop* actions that encode persistence. In the SAT model of a planning problem time-stamped propositional atoms (or variables) represent the action and facts of the problem. An atom $A(T)$, where A is an action, corresponds to the decision of whether

action A is taken or not at time T, and analogously for variables of the form $f(T)$ where f is a fact.

2. *Action based encoding*. In the SAT model of a planning problem time-stamped propositional atoms (or variables) from the planning graph represent the action of the problem. As in the direct encoding, an atom $A(T)$, where A is an action (including *noop* actions as in direct encoding), corresponds to the decision of whether action A is taken or not at time T.

3. *Indirect encoding*. In this category the variables are not directly mapped to a time-stamped fact or action. In the split action model of [106, 104] each time-stamped fact is mapped to variable, but each time-stamped action is presented as a conjunction of more than one variables. Moreover, there are auxiliary variables. In the SASE encoding [60, 61, 62] each time-stamped action is mapped to variable, but there is a variable mapped to a time-stamped legal transition from one fact to another, both being values of the same multi-valued variable of the SAS^+ representation of the problem. Auxiliary variables are also used in this encoding in order to reduce the number of clauses.

4.2.1 Direct and Action based encodings

In the *direct encoding* the variables correspond to time stamped facts $f(T)$ and time stamped actions $A(T)$, whereas in *action based encoding* the variables correspond *only* to time stamped actions $A(T)$. More precisely, in direct encodings the variables that are added emerge from the three rules below, whereas in action based encodings the variables emerge only from the second rule. Initial and final state are implicitly encoded in action based encodings.

1. Unit clauses for the initial and final state.

2. An action variable $A(T)$ is added to the theory if for each $p \in pre(A)$, p exists in the planning graph in layer T , and there is no pair $p_1, p_2 \in pre(A)$ s.t. p_1 and p_2 are marked as mutex in the layer T of the planning graph.

3. A proposition variable $p(T)$ is added to the theory if some action variable $A(T-1)$ is in the theory, with $p \in add(A)$.

To facilitate our study we first introduce a new encoding called *Graphplan-direct*, that is direct translation of the planning graph structure into propositional logic. All the *direct encodings* that we investigate in the rest of this work are subsets of the clause set of the Graphplan-direct formulation.

The clauses of the Graphplan-direct encoding are the following (1-8).

1. Unit clauses for the initial and final state.

2. $A(T) \rightarrow f(T)$, for every action A and fact f s.t. $f \in pre(A)$.

3. $A(T) \rightarrow f(T+1)$, for every action A and fact f s.t. $f \in add(A)$.

4. $A(T) \rightarrow \neg f(T+1)$, for every action A and fact f s.t. $f \in del(A)$.

5. $f(T) \rightarrow A_1(T-1) \vee \ldots \vee A_m(T-1)$, for every fact f and all actions A_i, $1 \leq i \leq m$ (including the noops) s.t. $f \in add(A_i)$.

6. $\neg f(T) \rightarrow A_1(T-1) \vee \ldots \vee A_m(T-1) \vee \neg f(T-1)$, for every fact f and all actions A_i, $1 \leq i \leq m$ s.t. $f \in del(A_i)$.

7.1 $\neg A_1(T) \vee \neg A_2(T)$, for every pair of actions A_1, A_2 such that the set $del(A_1) \cap pre(A_2)$ is non-empty.

7.2 $\neg A_1(T) \vee \neg A_2(T)$, for every pair of actions A_1, A_2 such that the set $del(A_1) \cap add(A_2)$ is non-empty.

7.3 $\neg A_1(T) \vee \neg A_2(T)$, if there is a pair of facts $f_1 \in pre(A_1)$, $f_2 \in pre(A_2)$ such that f_1, f_2 are mutually exclusive at time T.

8 $\neg f_1(T) \vee \neg f_2(T)$, for every pair of facts f_1, f_2 that are mutex at time T.

The set of clauses below (9) is used *only* in the *action based* encodings. All the action based encodings presented in this work are subset of clauses 7.1,7.2,7.3 and 9

9 $A(T) \rightarrow A_1(T-1) \vee \ldots \vee A_m(T-1)$, for every fact f such that $f \in pre(A)$ and all actions A_i, $1 \leq i \leq m$ (including the noops) s.t. $f \in add(A_i)$.

The first system that employed information derived from the planning graph in the propositional encoding of a planning problem was BLACKBOX [75]. BLACKBOX (version 43) supports different encodings, three of which we investigate here and denoted by BB-7, BB-31, and BB-32. Each of them is obtained by selecting the appropriate value (7, 31, or 32) of parameter *axioms*. The set of clauses of each of these encodings (which is a subset of the clauses of the Graphplan-direct model) is the following.

1. **BB-7**: Clauses 1, 2, 5, 7.1, 7.2, 7.3
2. **BB-31**: Clauses 1, 2, 3, 4, 5, 7.1, 8
3. **BB-32**: Clauses 1, 2, 3, 4, 5, 7.1, 7.2, 7.3, 8

Similarly to BLACKBOX, SATPLAN06 also supports different encodings. Two of them are direct encoding (mixed action/fact models), and the other two are action-based encodings (only actions). They are denoted by SATPLAN06-1, SATPLAN06-2, SATPLAN06-3 and SATPLAN06-4, and are obtained by setting the *encoding* parameter to value 1 to 4 respectively. Each of them contains the following clauses (again numbers refer to the Graphplan-direct model). The first two are direct encoding and the last two action based.

1. **SATPLAN06-4**: Clauses 1, 2, 5, 7.1, 7.2, 8
2. **SATPLAN06-3**: Clauses 1, 2, 5, 7.1, 7.2, 7.3, 8
3. **SATPLAN06-1**: Clauses 7.1, 7.2, 9
4. **SATPLAN06-2**: Clauses 7.1, 7.2, 7.3, 9

In action based encodings the initial facts and the goals are implicit encoded in the model. For example for encoding of a planning horizon T for each goal g of time T there is a clause $A(T-1) \vee \ldots \vee A_m(T-1)$ (including the noop) for all actions A_i, $1 \leq i \leq m$ (including the noops) s.t. $g \in add(A_i)$. Also for all actions in the first layer it holds that each of their preconditions are in the initial facts. The propositional theory that results from the above encodings, for a fixed number of time steps T_{max}, is given as input to a SAT solver. Any time step $T \leq T_{max}$, is a *valid time point*.

4.2.2 Long-Distance Mutual Exclusion

The mutual exclusion information that is derived from the planning graph in the previous encodings concerns facts or actions that refer to the same time, i.e. are binary clauses of the form $\neg a(T) \vee \neg b(T)$. This information has been generalized in [30], where binary clauses on variables that refer to different time points, i.e. binary clauses $\neg a(T) \vee \neg b(T+k)$, were introduced in the SAT model.

The Long-Distance Mutual Exclusion (londex) method of [30], is based on the state variable representation or *multi-valued domain formulation* (MDF), presented in section 3.1.2 of chapter 3, in which a planning domain is defined over a set $X = (x_1, \ldots, x_n)$ of multi-valued variables, where each x_i has an associated finite domain $\mathcal{D}_i$. If x is a multi-valued variable from X and v a value from its domain, $x = v$ denotes the assignment of v to x. To associate such an assignment $x = v$ with a boolean fact f, we use the notation $f = MDF(x, v)$.

For every multi-valued variable in a planning problem, the method of [30] builds the domain transition graph (DTG), from which the fact distances are computed. We call $londex_1$ this type of londexes between values of a MDF variable, that are computed using only the DTG of that variable. In [29] stronger londexes are computed with the use of a combination of DTGs. We call this type of londexes $londex_m$. The central notion of londexes is that of the distance between two facts f_1, f_2 in a DTG G_x, denoted by $\Delta_{G_x}(f_1, f_2)$.

Definition 27 Given a DTG G_x, the distance from a fact $f_1 = MDF(x, v_1)$ to another fact $f_2 = MDF(x, v_2)$, denoted by $\Delta_{G_x}(f_1, f_2)$, is defined as the minimum distance from vertex v_1 to vertex v_2 in G_x.

4.2.2.1 Long-Distance Mutual Exclusion on a single DTG: $londex_1$

Based on fact distances, $londex_1$ constraints for facts and actions are derived. In the following, $t(f)$ denotes the time step at which fact f is true, and $t(a)$ the time step at which an action is chosen. Moreover, we say that an action a is associated with a fact f if f appears in $pre(a)$, $add(a)$ or $del(a)$.

Definition 28 (Fact Londex) Given two boolean facts f_1 and f_2, that correspond to two nodes in a DTG G_x, such that $\Delta_{G_x}(f_1, f_2) = r$, then there is no valid plan in which $0 \leq t(f_2) - t(f_1) < r$.

There are two classes of actions londex constraints that are defined below.

Definition 29 (Class A Action Londex) If actions a and b are associated with a fact f, they are mutually exclusive if one of the following holds:

1. $f \in add(a), f \in del(b)$, and $t(a) = t(b)$

2. $f \in del(a), f \in pre(b)$, and $0 \leq t(b) - t(a) \leq 1$

Definition 30 (Class B Action Londex) If action a is associated with fact f_1 and action b with fact f_2, and it is invalid to have $0 \leq t(f_2) - t(f_1) < r$ according to fact londex constraints, then a and b are mutually exclusive if one of the following holds:

1. $f \in add(a), f \in add(b)$, and $0 \leq t(b) - t(a) \leq r - 1$

2. $f \in add(a), f \in pre(b)$, and $0 \leq t(b) - t(a) \leq r$

3. $f \in pre(a), f \in add(b)$, and $0 \leq t(b) - t(a) \leq r - 2$

4. $f \in pre(a), f \in pre(b)$, and $0 \leq t(b) - t(a) \leq r - 1$

4.2.2.2 Long-Distance Mutual Exclusion on multiple DTGs: $londex_m$

In [29], Chen et al. extend the idea of londexes. Instead of using only the DTG that contains the two facts f_1 and f_2 in order to compute the minimum distance between them, they use multiple DTGs in an attempt to find possibly stronger $londex_m$ constraints. The idea is that some of the facts that are preconditions in actions of the shortest(s) path(s) from f_1 to f_2 in their DTG graph G_1 may belong to a different DTG graph G_2, and the distance between them in G_2 imposes a minimum distance strictly greater than the one enforced by G_1 in the $londex1$ definition. In this case graph G_1 *depends* on G_2 because there is an action a that enables a transition in G_1 using a precondition from G_2.

For example assume f_1 and f_2 in G_1 with $\Delta_{G_1}(f_1, f_2) = 3$ and let their shortest path be $\xi = f_1 \longrightarrow v_1 \longrightarrow v_2 \longrightarrow f_2$. Suppose that the set of actions that change value from f_1 to v_1 have the common precondition p, whereas from v_2 to f_2 have the common precondition p', and both belong to a DTG graph G_2 and it holds that $\Delta_{G_2}(p, p') = 4$. Obviously, the minimum distance from f_1 to f_2 can be soundly updated to $\gamma(f_1, f_2) = min(max(\Delta_{G_1}(f_1, f_2), \Delta_{G_2}(p, p') + 1)) = 5$. In case G_2 depends on another DTG G_3, this can be used in the same manner in G_2, possibly leading to a stronger constraint for the distance between f_1 and f_2. For a transition from v to w in a DTG w (v) is a *successor* (*predecessor*) of v (w). The set of *successors* and *predecessors* of v is denoted by $succ(v)$ and $pred(v)$ respectively. Moreover, the set of shared preconditions of actions of the transition from v to w is denoted by $\mathcal{P}(v, w)$. Function γ, for f_1 and f_2, is defined recursively as:

$$\gamma(f_1, f_2) = \min_{v \in succ(f_1), w \in pred(f_2)} \{\max(\max_{p \in \mathcal{P}(f_1, v), p' \in \mathcal{P}(w, f_2)} \{\gamma(p, p')\} + 1, \Delta_G(v, w) + 2)\}$$

To capture the relation between the DTG graphs in the computation of γ function, the method of [29] employs the *Causal Graph* along with a cycle breaking mechanism, as *Causal Dependency Graph* is not acyclic in the general case.

In order to find more $londex_m$ constraints, distances are enhanced with *bridge analysis*. For instance, assume that in the above example there exist facts f_1', f_2' in G_1 such that any path in G_1 from f_1' to f_2' visits f_1 and f_2 in this order. The pair of facts f_1, f_2 is a *bridge pair* (hence the name) for the pair of facts f_1' and f_2'. The minimum distance from f_1' to f_2' can be soundly updated (improved) to: $\Delta_{G_1}(f_1', f_1) + \gamma(f_1, f_2) + \Delta_{G_1}(f_2, f_2')$.

The distances found by the calculation of γ function and the *bridge analysis* are used to derive fact and action $londex_m$ constraints in a similar manner as was described previously in subsection 4.2.2.1 for facts and actions. Due to the large number of $londex_m$ clauses Chen et al. [29] alter the unit propagation mechanism of the SAT solver to create on the fly only the $londex_m$ constraints that are needed for unit propagation, instead of creating all $londex_m$ clauses before search and adding them in the SAT theory. Experiments on planning benchmarks revealed better solution times when using $londex_m$ (than only $londex_1$ and no londex at all) for SATPLAN04 and SATPLAN06 encodings.

4.2.3 Indirect encodings. SASE and SOLE planners

Two *indirect encoding* schemes for parallel (step-optimal) planning are the SAS+ based encoding *SASE* of Huang, Chen and Zhang [60, 61, 62] in the SASE planner and the *split action encoding* of the *SOLE* planner of Robinson et al. [106, 104]. Both planners follow the solve and expand method of the SATPLAN framework.

SASE [60, 61, 62] encoding is based on state variables representation. The variables of the SASE encoding are either actions for a time step (layer) of the plan (as in direct and action encodings) or *transitions* between values for each state variable of the problem for a time step. Huang et al. used several methods to reduce the size of the encoding, exploiting that variables representing actions of the same transition, as well as transitions of the same state variable, are *cliques*. The trivial encoding of n clique variables uses $O(n^2)$ (mutex binary) clauses. They used the compact representation of Rintanen [98] instead, that encodes a clique of n variables using $O(n.\log_2 n)$ auxiliary variables but only $O(n.\log_2 n)$ binary clauses (instead of $O(n^2)$). They also omit the addition of clauses representing an action clique for a transition if is a subset of another clique for another transition as redundant. For any transition that can be made true only by one action the variable of action is replaced with the one of its transitions in the encoding since are logically equivalent. Our experimental results show that `SMP` planner performs better than `SASE`.

The idea of indirect representations of actions in order to reduce the number of action variables is not new. In fact a split action representation was implemented by Kautz and Selman [72] in their first attempt to solve planning as a satisfiability problem (for sequential planning). The major difficulty in split action representations of actions arises in parallel planning where many (not mutex) actions may be executed in a single time step. The first split action encoding for parallel step-optimal planning is introduced by Robinson et al. [106, 104] in the *SOLE* planner.

SOLE planner is the extension of an older work of the same authors [105] where they present an encoding in *PARA-L* planner. Their encoding in [105] permits parallel action execution only if interference between actions does not occur.

The variables in the SAT compilation in [106, 104] are time stamped facts (as in *direct encoding*) and time stamped ground conditions for actions as they emerge from a planning graph for a planning horizon. An action is true in a time step if all the variables representing the ground conditions of the action at that time step are true. There are also *auxiliary* (time stamped) variables that are used to represent action mutexes of the same operator. Additional *auxiliary* (time stamped) variables are used in auxiliary clauses to ensure that whole instances of operators of the problem are executed. The authors did not provide publicly any binaries or source code therefore we did not perform any experiments for *SOLE* planner.

4.3 The relative strength of the encodings

In this section we investigate the relative constraint propagation strength of encodings defined on the same variables. More specifically, we compare the relative constraint propagation power of `BLACKBOX` and the direct `SATPLAN06` encodings with respect to each other, and then we compare the two action based encodings of `SATPLAN06` encodings.

4.3.1 Comparing encodings

As noted earlier, almost all state-of-the-art SAT solvers employ Unit propagation for constraint propagation. In the following, $UP(T)$ denotes the closure of theory T under Unit Propagation. The notion of *UP-redundancy* plays a central role in our analysis, and is defined as follows.

Definition 31 The binary clause $l_1 \vee l_2$ is *UP-redundant* wrt a theory T iff either $l_1 \vee l_2 \in T$ or $l_j \in UP(T \cup \{\neg l_i\})$, for $i \neq j$ and $i, j \in \{1, 2\}$.

Below, several notions regarding the relative strength of propositional theories wrt binary clauses are defined.

Definition 32 Theory T_1 is at *least as strong* as theory T_2 wrt UP and binary clauses, denoted by $T_1 \geq_{UP} T_2$, iff every clause of $T_2 \backslash T_1$ is binary and UP-redundant wrt T_1.
Theory T_1 is *strictly stronger* than theory T_2 wrt UP and binary clauses, denoted by $T_1 >_{SUP} T_2$, iff $T_1 \geq_{UP} T_2$ or $T_1 \supset T_2$ and $T_2 \not\geq_{UP} T_1$.
Theory T_1 is *more compact* than theory T_2 wrt UP and binary clauses, denoted by $T_1 >_C T_2$, iff $T_1 \geq_{UP} T_2$ and $T_1 \subset T_2$.

4.3.2 Comparison of direct encodings

It is easy to see that the encodings are related as follows: for any (STRIPS) planning problem P, SATPLAN06-4(P) $\subset$ SATPLAN06-3(P) $\subset$ BB-32(P), and BB-7(P) $\subset$ SATPLAN06-3(P). The following proposition shows that some of the mutex clauses are UP-redundant in some encodings.

Proposition 1 The set of clauses 7.3 is UP-redundant wrt any propositional encoding that contains the set of clauses 2 and 8.

Proof Let $\neg A_1(T) \vee \neg A_2(T)$ be a clause with A_1 and A_2 two actions such that there is a pair of facts $f_1 \in pre(A_1)$, $f_2 \in pre(A_2)$ such that f_1, f_2 are mutually exclusive at level T. We will show that $\neg A_2(T) \in UP(T_P \cup \{A_1(T)\})$. From $A_1(T)$ and clause $A_1(T) \rightarrow f_1(T)$ we obtain $f_1(T)$. Since the theory contains axioms 8, it must contain the clause $\neg f_1(T) \vee \neg f_2(T)$. From this clause and $f_1(T)$ we obtain $\neg f_2(T)$, form which, together with $A_2(T) \rightarrow f_2(T)$ we conclude $\neg A_2(T)$. The proof of $\neg A_1(T) \in UP(T_P \cup \{A_2(T)\})$ is symmetric. ■

A direct consequence of the above proposition is the following relation between the two direct `SATPLAN06` encodings.

Corollary 1 For any planning problem P, SATPLAN06-4$(P) >_C$ SATPLAN06-3(P).

Another result that is stated formally below can be used to simplify theories that contain clause sets 3 and 4.

Proposition 2 The set of clauses 7.2 is UP-redundant wrt any propositional encoding that contains the set of clauses 3 and 4.

Proof Let $\neg A_1(T) \vee \neg A_2(T)$ be a clause with A_1 and A_2 two actions such that at least one of the sets $del(A_1) \cap add(A_2)$ and $del(A_2) \cap add(A_1)$ is non-empty. Assume that $del(A_1) \cap add(A_2) \neq \emptyset$ (the other case is symmetric) and let $f \in del(A_1) \cap add(A_2)$. We will show that $\neg A_2(T) \in UP(T_P \cup \{A_1(T)\})$. The theory contains a clause of the form $A_1(T) \rightarrow \neg f(T+1)$ from which $\neg f(T+1)$ is derived. From this and clause $A_2(T) \rightarrow f(T+1)$, $\neg A_2(T)$ is concluded. ■

A direct consequence of the above proposition is that encoding BB-32 can be simplified by removing clauses 7.2. Similarly, by proposition 1, clauses 7.3 can also be omitted. Therefore, the following corollary is immediate.

Corollary 2 For any planning problem P, BB-31$(P) >_C$ BB-32(P).

A similar observation holds for the Graphplan-direct encoding. By removing the UP-redundant clauses 7.2 and 7.3 we obtain SATPLAN^{max} encoding which is a direct encoding and contains the following clauses:

- **SATPLAN^{max}**: Clauses 1, 2, 3, 4, 5, 6, 7.1, 8.

On the other hand, the set of clauses 8 *is not UP-redundant wrt to any encoding* that contains any of the other clauses (i.e. 1 to 7.3). From this we conclude that, for all problems P, BB-31(P) $>_{SUP}$ BB-7(P). Similarly clause sets 3 and 4 are not UP-redundant wrt to any other clause, and therefore, BB-31(P) $>_{SUP}$ SATPLAN06-4(P). Hence, it seems that from the implemented encodings of planning as satisfiability, BB-31 is the strongest. Finally, SATPLAN$^{max}(P)$ $>_{SUP}$ BB-31(P), for any problem P, due to the existence of clause set 6. We collect these results in the following corollary.

Corollary 3 For any planning problem P,

- BB-31(P) $>_{SUP}$ BB-7(P)
- BB-31(P) $>_{SUP}$ SATPLAN06-4(P)
- SATPLAN$^{max}(P)$ $>_{SUP}$ BB-31(P)

Note, that the SATPLAN^{max} encoding uses only one set of mutex actions, namely set 7.1. However, it is possible that a clause is included in several sets of mutex clauses, each for a different reason. Therefore, a mutex pair of actions that belongs to set 7.1 may also belong to other mutex sets that are UP-redundant. The size of clause set 7.1 of SATPLAN^{max}, can be reduced by omitting all clauses of this set that also belong to sets 7.2 or 7.3. Furthermore, all mutex action clauses on actions A_1 and A_2 and time T that contain add effects p_1 and p_2 respectively such that p_1 and p_2 are mutex at time $T + 1$ can also be omitted. We call the resulting encoding `SAT-MAX-PLAN`, or `SMP` for short.

4.3.3 Comparison of action based encodings

It is easy to see that the encodings are related as follows: for any planning problem P, SATPLAN06-1(P) $\subset$ SATPLAN06-2(P). We prove via an example that SATPLAN06-2(P) is a stronger encoding than SATPLAN06-1(P).

Consider a planning domain P with an action $mv(i, j)$ that moves an object from location i to location j if $|i-j| = 1$, and assume locations $1 \leq l \leq 3$. Assume that in the initial state the object is at location 1. The sets of clauses of SATPLAN06-1(P) and SATPLAN06-2(P) are as follows.

To represent the initial state, we assume the dummy time point -1 for which all actions are false except $noop(1, -1)$. The sets of clauses associated with P are the following. Note that if a clause exists in two types of clauses is presented only once. For example the clause $\neg mv(2, 3, T) \vee \neg mv(1, 2, T)$ is in 7.2 and 7.3

Clauses 7.1

$\neg mv(1, 2, T) \vee \neg noop(1, T)$, $\neg mv(2, 3, T) \vee \neg noop(2, T)$

$\neg mv(2, 1, T) \vee \neg noop(2, T)$, $\neg mv(3, 2, T) \vee \neg noop(3, T)$

$\neg mv(2, 3, T) \vee \neg mv(2, 1, T)$

Clauses 7.2

$\neg mv(1, 2, T) \vee \neg mv(2, 3, T)$

$\neg mv(1, 2, T) \vee \neg mv(2, 1, T)$, $\neg mv(2, 3, T) \vee \neg mv(3, 2, T)$

Clauses 7.3

$\neg mv(3, 2, T) \vee \neg mv(1, 2, T)$

$\neg mv(2, 1, T) \vee \neg mv(3, 2, T)$, $\neg mv(1, 2, T) \vee \neg noop(2, T)$

$\neg mv(1, 2, T) \vee \neg noop(3, T)$, $\neg mv(2, 1, T) \vee \neg noop(1, T)$

$\neg mv(2, 1, T) \vee \neg noop(3, T)$, $\neg mv(2, 3, T) \vee \neg noop(1, T)$

$\neg mv(2,3,T) \vee \neg noop(3,T)$, $\neg mv(3,2,T) \vee \neg noop(1,T)$

$\neg mv(3,2,T) \vee \neg noop(2,T)$, $\neg noop(1,T) \vee \neg noop(2,T)$

$\neg noop(1,T) \vee \neg noop(3,T)$, $\neg noop(2,T) \vee \neg noop(3,T)$

Clauses 9 for $T \geq 0$

$\neg mv(1,2,T) \vee mv(2,1,T-1) \vee noop(1,T-1)$

$\neg mv(2,1,T) \vee mv(1,2,T-1) \vee mv(3,2,T-1) \vee noop(2,T-1)$

$\neg mv(2,3,T) \vee mv(1,2,T-1) \vee mv(3,2,T-1) \vee noop(2,T-1)$

$\neg mv(3,2,T) \vee mv(2,3,T-1) \vee noop(3,T-1)$

$\neg noop(i,T) \vee noop(i,T-1) \vee mv(i-1,i,T-1) \vee mv(i+1,i,T-1)$ for valid values of i

Assume now the theory T_P^{06-1} of SATPLAN06-1 encoding and the action $mv(2,1,3)$. It is easy to see that $UP(T_P^{06-1} \cup \{mv(2,1,3)\})$ includes the literals that are derived by propagation form the initial state

$\neg mv(2,1,0), \neg mv(2,3,0), \neg mv(3,2,0)\ \neg noop(2,0), \neg noop(3,0)$ from (9)

$\neg mv(3,2,1), \neg noop(3,1)$ from (9)

From $mv(2,1,3)$ we derive $\neg noop(2,3)$ and $\neg mv(2,3,3)$ via the rules 7.1 and $\neg mv(1,2,3)$ from 7.2. No other inference is possible. Note however that for theory T_P^{06-2} of SATPLAN06-2 encoding and $S = \{\neg mv(3,2,3), \neg noop(1,3), \neg noop(3,3)\}$ it holds that $S \subseteq UP(T_P^{06-2} \cup \{mv(2,1,3)\})$, whereas $S \cap UP(T_P^{06-2} \cup \{mv(2,1,3)\}) = \emptyset$. This leads to the following result.

Proposition 3 The set of clauses 7.3 is *not UP-redundant* wrt encodings on the set of clauses 7.1, 7.2 and 9.

A direct consequence of the above proposition is the following relation between the two action `SATPLAN06` encodings.

Corollary 4 For any planning problem P, SATPLAN06-2$(P) >_{SUP}$ SATPLAN06-1(P).

Although in SATPLAN06-2 encoding of the problem can be derived better constraint propagation than in SATPLAN06-1 through unit propagation, the run-times tell a very different story. Both encodings were implemented and experimentally tested in the predecessor of SATPLAN06, the SATPLAN04. In that work, where SATPLAN06-1 and SATPLAN06-2 are called ”skinny action-based encoding” and ”non-skinny action-based encoding” respectively, SATPLAN06-1 proved to be much faster than SATPLAN06-2, due to large size of the theories encoded in SATPLAN06-2 encoding. Moreover SATPLAN06-2 caused memory problems in large problems from many domains due to the large number of 7.3 action mutexes.

4.4 Londex Propagation in Propositional Planning

From the STRIPS encoding P of a planning problem, we can construct its state variable representation P_M, using a translation M as those described e.g. in [54]. For each state variable (or multi-valued variable) X of P_M with domain $\mathcal{D}_X$, we denote by $X(v)$ the fact in its STRIPS representation P that corresponds to the assignment of value $v \in \mathcal{D}_X$ to variable X. Moreover, $X(v, T)$ denotes the atom (in the planning graph and the propositional theory) that represents the truth value of $X(v)$ at time T. In order to abstract away from the details of the particular method that is used to construct the multi-valued representation of a STRIPS domain, and therefore simplify our discussion, we make some, we believe, natural assumptions about the domains we consider.

Definition 33 A multi-valued translation method M that translates STRIPS problems into their multi-value representation, satisfies the *domain compatibility assumption* if for every STRIPS problem P and its multi-valued representation P_M the following conditions hold:

1. Let X be a multi-valued variable of P_M with domain $\mathcal{D}_X$ and A any action of P. If $X(v_i) \in add(A)$ for $v_i \in \mathcal{D}_X$, then $X(v_j) \in del(A) \cap pre(A)$ for some $v_j \in \mathcal{D}_X$ with $i \neq j$.

2. If X is a multi-valued variable of P_M with domain $\mathcal{D}_X$, then the initial state assigns true to exactly one fact of the form $X(v_i)$ for $v_i \in \mathcal{D}_X$.

We can now prove that for translations that satisfy the domain compatibility assumption, Graphplan marks as mutex all facts that refer to the different values of a multi-valued variable.

Proposition 4 Let P_M be the translation of a STRIPS problem P under a translation method M that satisfies the domain compatibility assumption. If X is a multi-valued variable of P_M with domain $\mathcal{D}_X$, in the planning graph all pairs of facts of the from $X(v_i), X(v_j)$, with $v_i, v_j \in \mathcal{D}_X$ and $i \neq j$, are mutex in all its levels where they both appear.

Proof We prove the claim inductively on planning graph levels.

Base case. Assume that both $X(v_i)$ and $X(v_j)$ appear on (fact) level 1 of the planning graph. We prove that they are marked as mutually exclusive by Graphplan. Suppose first that one of $X(v_i)$ and $X(v_j)$, say $X(v_i)$, appears in the initial state (fact level 0). Since $X(v_j)$ appears on level 1, there must be some actions $A_1^{v_j}, \ldots, A_n^{v_j}$ such that $X(v_j) \in add(A_c^{v_j})$ and $X(v_i) \in del(A_c^{v_j}) \cap pre(A_c^{v_j})$, for $1 \leq c \leq n$. On the other hand, $X(v_i)$ appears on level 1 because of $noopX(v_i)$. Observe that $noopX(v_i)$ is mutex with all actions $A_c^{v_j}$ in the preceding action level, and therefore $X(v_i)$ and $X(v_j))$ are marked as mutex at level 1.

Assume now that $X(v_t)$ appears in the initial state, and therefore, by the domain compatibility assumption, none of $X(v_i)$ and $X(v_j)$ does. Then, there must be two sets of actions, $A_1^{v_j}, \ldots, A_n^{v_j}$ and $A_1^{v_i}, \ldots, A_m^{v_i}$, such that $X(v_j) \in add(A_c^{v_j})$, $X(v_t) \in del(A_c^{v_j}) \cap pre(A_c^{v_j})$, for $1 \leq c \leq n$, and $X(v_i) \in add(A_d^{v_i})$, $X(v_t) \in del(A_d^{v_i}) \cap pre(A_d^{v_i})$, for $1 \leq d \leq m$. Observe that every action

$A_c^{v_j}$ deletes $X(v_t)$ which is a precondition of all actions $A_d^{v_i}$. Therefore, every action $A_c^{v_j}$ is mutex with every action $A_d^{v_i}$. Hence, $X(v_i)$ and $X(v_j)$ are also marked as mutex at fact level 1.

Inductive hypothesis. Assume that for some planning graph level k, Graphplan marks as mutex all pairs of facts of the form $X(v_i), X(v_j)$, with $v_i, v_j \in \mathcal{D}_X$.

Inductive step. We prove that the same holds for graph level $k+1$ for all pairs of facts of the form $X(v_i), X(v_j)$, with $v_i, v_j \in \mathcal{D}_X$. Let $A_c^{v_i}$ be an action such that $X(v_i) \in add(A_c^{v_i})$, and $A_d^{v_j}$ an action such that $X(v_j) \in add(A_d^{v_j})$.

First assume that $A_c^{v_i}$ and $A_d^{v_j}$ are the Noop actions $Noop_{X(v_i)}$ and $Noop_{X(v_j)}$ respectively. Obviously $X(v_i) \in pre(Noop_{X(v_i)})$ and $X(v_j) \in pre(Noop_{X(v_j)})$. By the inductive hypothesis $X(v_i)$ and $X(v_j)$ are marked as mutex at layer k, hence actions $Noop_{X(v_i)}$ and $Noop_{X(v_j)}$ are marked as mutex at layer k (since their preconditions are mutex). Now assume that only one of the actions $A_c^{v_i}$ and $A_d^{v_j}$ is a $Noop$. Without loss of generality let $A_c^{v_i} = Noop_{X(v_i)}$. By the domain compatibility assumption, there is a fact $X(v_d) \in del(A_d^{v_j}) \cap pre(A_d^{v_j})$, with $v_d \in \mathcal{D}_X$. Assume first that $v_i \neq v_d$. From the inductive hypothesis we know that $X(v_i), X(v_d)$ are mutex at fact level k, therefore actions $Noop_{X(v_i)}$ and $A_d^{v_j}$ are mutex at action level k. if $v_i = v_d$, then $pre(Noop_{X(v_i)}) \cap del(A_d^{v_j}) = \{X(v_i)\}$, therefore $Noop_{X(v_i)}$ and $A_d^{v_j}$ are again mutex since $pre(Noop_{X(v_i)}) \cap del(A_d^{v_j}) \neq \emptyset$.

Now assume that neither of the actions $A_c^{v_i}$ and $A_d^{v_j}$ is a Noop. By the domain compatibility assumption, there are facts $X(v_c) \in del(A_c^{v_i}) \cap pre(A_c^{v_i})$, $X(v_d) \in del(A_d^{v_j}) \cap pre(A_d^{v_j})$, with $v_c, v_d \in \mathcal{D}_X$. Assume first that $v_c \neq v_d$. From the inductive hypothesis we know that $X(v_c), X(v_d)$ are mutex at fact level k, therefore actions $A_c^{v_i}$ and $A_d^{v_j}$ are mutex at action level k. On the other hand, if $v_c = v_d$, then $pre(A_c^{v_i}) \cap del(A_d^{v_j}) \neq \emptyset$, therefore $A_c^{v_i}$ and $A_d^{v_j}$ are again mutex. Therefore, any pair of actions $A_c^{v_i}$ and $A_d^{v_j}$ that adds $X(v_i)$ and $X(v_j)$ respectively is mutex at level k. Hence, $X(v_i), X(v_j)$ is marked as mutex at level $k+1$. ■

A similar result is proven for actions that have multi-valued variables in their add effects.

Proposition 5 Let P_M be the translation of a STRIPS problem P under a translation method M that satisfies the domain compatibility assumption. If X is a multi-valued variable of P_M with domain $\mathcal{D}_X$, in the planning graph all pairs of action A_i, A_j (including noops) such that $X(v_i) \in add(A_i)$, $X(v_j) \in add(A_j)$, for $v_i, v_j \in \mathcal{D}_X$, are mutex in all its levels where they both appear.

Proof We prove that A_i, A_j are marked mutually exclusive at any planning graph level k where they both appear. First assume that both A_i, A_j are Noops. Then $A_i = Noop(X(v_i))$ and $A_j = Noop(X(v_j))$. It holds that $pre(Noop(X(v_i))) = \{X(v_i)\}$ and $pre(Noop(X(v_j))) = \{X(v_j)\}$. By proposition 4 $X(v_i)$ and $X(v_j)$ are marked as mutex at all layers, hence $Noop(X(v_i))$ and $Noop(X(v_j))$ are also marked as mutex at all layers because they have mutex preconditions.

Now assume that at least one of A_i, A_j is not a Noop. Let $X(v_i^p) \in pre(A_i)$ and $X(v_j^p) \in pre(A_j)$, with $v_i^p, v_j^p \in \mathcal{D}_X$. Assume first that $v_i^p = v_j^p$. Then, if none of A_i, A_j is a noop, $X(v_i^p) \in del(A_i) \cap del(A_j)$, and therefore each of these actions deletes the precondition of the other. Hence they will be marked as mutex at all levels. If one of these action, say A_i, is a noop, then again A_j deletes its preconditions, and therefore again A_i, A_j are mutex. Assume now that $v_i^p \neq v_j^p$. By proposition 4 we know that $X(v_i^p)$ and $X(v_j^p)$ are marked mutually exclusive at all level they both appear, therefore A_i, A_j are also marked as mutex. ■

In the rest of the chapter we assume that londex constraints are generated from the multi-valued representation of a planning domain by a translation method that satisfies the domain compatibility assumption. Moreover, we assume that londex constraints are translated into clauses in a straight-forward manner, i.e. a Class A action londex on actions A_1 and A_2 translates into a set of binary clauses $\neg A_1(T) \vee \neg A_2(T+1)$ for all valid time points.

4.4.1 $londex_1$ propagation in SATPLAN06

In the following we analyze the effects of various londex constraints on the constraint propagation of a UP based SAT solver. The notions of forward and backward redundancy that are defined below are central in our analysis.

Definition 34 A clause of the form $\neg p(T) \vee \neg q(T+k)$ that corresponds to a londex constraint of a planning problem P is *forward UP-redundant* wrt to an encoding T_P of P if $\neg q(T+k) \in UP(T_P \cup \{p(T)\})$. Similarly, the clause is *backward UP-redundant* wrt to T_P if $\neg p(T) \in UP(T_P \cup \{q(T+k)\})$.

We start by analyzing the effects of londex constraints of Class A. This class contains constraints that refer to actions that cannot be executed in parallel, as well as constraints that relate actions that are one time step apart. Note that action mutexes that refer to the same time point are included in `SATPLAN06` encoding, therefore we do not consider them. For the other type of Class A londex constraints, we show below that they are forward UP-redundant wrt the SATPLAN06-4 model, which we refer to as `SATPLAN06` encoding.

Proposition 6 Let A_1, A_2 be actions (including noops) and f a fact of a planning problem P such that $f \in del(A_1)$ and $f \in pre(A_2)$. The set of clauses $\neg A_1(T) \vee \neg A_2(T+1)$, for all valid points T, is forward UP-redundant wrt the `SATPLAN06` encoding of the problem.

Proof Let T_P be the SATPLAN06 encoding of a problem P. We prove that $\neg A_2(T+1) \in UP(T_P \cup \{A_1(T)\})$. Let $A_1^f, A_2^f, \ldots, A_k^f$ be the actions that contain f in their add effects (including noop). Theory T_P contains the clauses

1. $\neg A_2(T+1) \vee f(T+1)$

2. $\neg f(T+1) \vee A_1^f(T) \vee A_2^f(T) \vee \ldots \vee A_k^f(T)$

3. A set of binary clauses of the form $\neg A_1(T) \vee \neg A_i^f(T), 1 \leq i \leq k$.

From $A_1(T)$ and the set of clauses (3) above, UP derives the set of unit clauses $\neg A_i^f(T), 1 \leq i \leq k$ From these clauses and clause (2), UP entails the unit clause $\neg f(T+1)$ and from clause (1) $\neg A_2(T+1)$. Therefore $\neg A_2(T+1) \in UP(T \cup \{A_1(T)\})$.

■

We investigate now long distance constraints for facts, and show that they are also forward UP-redundant wrt the `SATPLAN06` encoding.

Proposition 7 Let X be a multi-valued variable of a planning problem P with domain $\mathcal{D}_X$, and T_P the SATPLAN06 encoding of P. Then, for any two values $v_i, v_j \in \mathcal{D}_X$ and $k \geq 0$ such that $\Delta_{G_X}(v_i, v_j) > k$, the set of clauses $\neg X(v_j, T+k) \vee \neg X(v_i, T)$ is forward UP-redundant wrt T_P for all valid time points T. Furthermore, for all actions A^j such that $X(v_j) \in pre(A^j)$, the set of clauses $\neg A^j(T+k) \vee \neg X(v_i, T)$ is also forward UP-redundant.

Proof We prove inductively on k that $\neg X(v_j, T+k) \in UP(T_P \cup \{X(v_i, T)\})$ for all v_j s.t. $\Delta_{G_X}(v_i, v_j) > k$. Moreover, we show within the same inductive proof, that $\neg A^j(T+k) \in UP(T_P \cup \{X(v_i, T)\})$ for all actions A^j such that $X(v_j) \in pre(A^j)$ and $\Delta_{G_X}(v_i, v_j) > k$.

Base case. We prove that the theorem holds for $k = 0$, that is, if $\Delta_{G_X}(v_i, v_j) > 0$, $\neg X(v_j, T) \in UP(T_P \cup \{X(v_i, T)\})$. First note that $\Delta_{G_X}(v_i, v_j) > 0$ holds for all $v_i, v_j \in \mathcal{D}_X$, $j \neq i$. By proposition 4, T_P contains the clauses $\neg X(v_i, T) \vee \neg X(v_j, T)$, for all $v_i, v_j \in \mathcal{D}_X$, $j \neq i$. Therefore, $\neg X(v_j, T) \in UP(T_P \cup \{X(v_i, T)\})$. Furthermore, if A^j is an action such that $X(v_j) \in pre(A^j)$, then T_P contains the clause $\neg A^j(T) \vee X(v_j, T)$. From this clause and $\neg X(v_j, T) \in UP(T_P \cup \{X(v_i, T)\})$, we conclude that $\neg A^j(T) \in UP(T_P \cup \{X(v_i, T)\})$.

Inductive hypothesis. Assume that for some $k \geq 0$, $\neg X(v_j, T+k) \in UP(T_P \cup \{X(v_i, T)\})$ holds for all facts $X(v_j)$ such that $\Delta_{G_X}(v_i, v_j) > k$. Furthermore, $\neg A^j(T+k) \in UP(T_P \cup \{X(v_i, T)\})$ for all actions A^j such that $X(v_j) \in pre(A^j)$ and $\Delta_{G_X}(v_i, v_j) > k$.

Inductive step. We prove first that $\neg X(v_j, T+k+1) \in UP(T_P \cup \{X(v_i, T)\})$ holds for all facts $X(v_j)$ such that $\Delta_{G_X}(v_i, v_j) > k+1$. Let $A^j_1, A^j_2, \ldots, A^j_m$ be the actions that have $X(v_j)$ in their add effects. Then T_P contains the clause

$\neg X(v_j, T+k+1) \vee A^j_1(T+k) \vee A^j_2(T+k) \vee \ldots \vee A^j_m(T+k) \vee noopX(v_j, T+k)$.

Since $\Delta_{G_X}(v_i, v_j) > k+1$, implies $\Delta_{G_X}(v_i, v_j) > k$, by the inductive hypothesis $\neg X(v_j, T+k) \in UP(T_P \cup \{X(v_i, T)\})$. From this and the binary clause $\neg noopX(v_j, T+k) \vee X(v_j, T+k)$ we conclude $\neg noopX(v_j, T+k) \in UP(T_P \cup \{X(v_i, T)\})$. Assume now that there is some action A^j_c, for $1 \leq c \leq m$, such that $\neg A^j_c(T+k) \notin UP(T_P \cup \{X(v_i, T)\})$, and let $X(v_b)$, $v_b \in \mathcal{D}_X$, be a precondition of A^j_c. Then, it can not be the case that $\Delta_{G_X}(v_i, v_b) > k$, because then, by the induction hypothesis, $\neg A^j_c(T+k) \in UP(T_P \cup \{X(v_i, T)\})$. Therefore, $\Delta_{G_X}(v_i, v_b) \leq k$. Then there must exist a path in G_X from v_i to v_b of length at most k, and an arc from v_b to v_j, therefore $\Delta_{G_X}(v_i, v_j) \leq k+1$. However, this contradicts the assumption $\Delta_{G_X}(v_i, v_j) > k+1$. Therefore, it must be the case that $\neg A^j_c(T+k) \in UP(T_P \cup \{X(v_i, T)\})$, for all $1 \leq c \leq m$. Hence, $\neg X(v_j, T+k+1) \in UP(T_P \cup \{X(v_i, T)\})$.

We now prove that $\neg A^j(T+k+1) \in UP(T_P \cup \{X(v_i, T)\})$ for all actions A^j such that $X(v_j) \in pre(A^j)$ and $\Delta_{G_X}(v_i, v_j) > k+1$. From the first part of the proof we know that $\neg X(v_j, T+k+1) \in UP(T_P \cup \{X(v_i, T)\})$. Moreover, theory T_P contains the clause $\neg A^j(T+k+1) \vee X(v_j, T+k+1)$. Therefore, $\neg A^j(T+k+1) \in UP(T_P \cup \{X(v_i, T)\})$. This completes the proof. ■

The results that follow show that all forms of Class B action londex constraints are forward UP-redundant. The proofs of these propositions give some insight into the propagation taking place in a UP-based SAT solver.

Proposition 8 Let X be a multi-valued variable of a planning problem P with domain $\mathcal{D}_X$, T_P the SATPLAN06 encoding of P, and $v_1, v_2 \in \mathcal{D}_X$ such that $\Delta_{G_X}(v_1, v_2) > k$. If A_1, A_2 are actions (including noops) such that $X(v_1) \in pre(A_1)$ and $X(v_2) \in pre(A_2)$, then the set of clauses $\neg A_2(T + k) \vee \neg A_1(T)$ is forward UP-redundant wrt to T_P for all valid time points.

Proof We show that $\neg A_2(T + k) \in UP(T_P \cup \{A_1(T)\})$. Theory T_P contains the clauses $\neg A_1(T) \vee X(v_1, T)$ and $\neg A_2(T+k) \vee X(v_2, T+k)$. Therefore, $X(v_1, T) \in UP(T_P \cup \{A_1(T)\})$. By proposition 7, $\neg X(v_2, T + k) \in UP(T_P \cup \{X(v_1, T)\}$, and therefore $\neg X(v_2, T + k) \in UP(T_P \cup \{A_1(T)\})$. By clause $\neg A_2(T+k) \vee X(v_2, T+k)$, we obtain $\neg A_2(T+k) \in UP(T_P \cup \{A_1(T)\})$. ■

Proposition 9 Let X be a multi-valued variable of a planning problem P with domain $\mathcal{D}_X$, T_P the SATPLAN06 encoding of P, and $v_1, v_2 \in \mathcal{D}_X$ such that $\Delta_{G_X}(v_1, v_2) > k$. If A_1, A_2 are actions (including noops) such that $X(v_1) \in add(A_1)$ and $X(v_2) \in add(A_2)$, then the set of clauses $\neg A_2(T + k) \vee \neg A_1(T)$ is forward UP-redundant wrt to T_P for all valid time points.

Proof We prove inductively on k that $\neg A_2(T+k) \in UP(T_P \cup \{A_1(T)\})$ for any pair of actions A_1, A_2 such that $X(v_1) \in add(A_1)$ and $X(v_2) \in add(A_2)$ and $\Delta_{G_X}(v_1, v_2) > k$.

Base case: We prove first the case $k = 0$. Note that $\Delta_{G_X}(v_1, v_2) > 0$ for all $v_1, v_2 \in \mathcal{D}_X$, $j \neq i$. Therefore, we must show that $\neg A_2(T) \in UP(T_P \cup \{A_1(T)\})$ for any pair of actions A_1, A_2 such that $X(v_1) \in add(A_1)$ and $X(v_2) \in add(A_2)$ with $v_1 \neq v_2$. Assume first that $X(v_p) \in pre(A_1) \cap pre(A_2)$ for $v_p \in \mathcal{D}_X$. Then $X(v_p) \in del(A_1) \cap del(A_2)$, therefore $\neg A_2(T) \vee \neg A_1(T) \in T_P$. Assume now that $X(v_p^1) \in pre(A_1)$ and $X(v_p^2) \in pre(A_2)$ with $v_p^1, v_p^2 \in \mathcal{D}_X$

and $v_p^1 \neq v_p^2$. Theory T_P contains the clauses $\neg A_2(T) \vee X(v_p^2, T)$ and $\neg A_1(T) \vee X(v_p^1, T)$. By proposition 4, it also contains the clause $\neg X(v_p^1, T) \vee \neg X(v_p^2, T)$. From these clauses it follows that $\neg A_2(T) \in UP(T_P \cup \{A_1(T)\})$.

Inductive hypothesis. Assume that for some $k \geq 0$, $\neg A_2(T + k) \in UP(T_P \cup \{A_1(T)\})$ holds for all pairs of actions A_1, A_2 such that $X(v_1) \in add(A_1)$ and $X(v_2) \in add(A_2)$ and $\Delta_{G_X}(v_1, v_2) > k$.

Inductive step: We prove that $\neg A_2(T + k + 1) \in UP(T_P \cup \{A_1(T)\})$ holds for all pairs of actions A_1, A_2 such that $X(v_1) \in add(A_1)$ and $X(v_2) \in add(A_2)$ and $\Delta_{G_X}(v_1, v_2) > k + 1$. Let $X(v_p^2) \in pre(A_2)$ with $v_p^2 \in \mathcal{D}_X$, and let $A_2^{p_c}$, $1 \leq c \leq n$, be the set of actions that have $X(v_p^2)$ in their add effects. Clearly, $\Delta_{G_X}(v_1, v_p^2) > k$. From the inductive hypothesis we know that $\neg A_2^{p_c}(T + k) \in UP(T_P \cup \{A_1(T)\})$ for all $1 \leq c \leq n$. Moreover, theory T_P contains the clause $\neg X(v_p^2, T + k + 1) \vee A_2^{p_1}(T + k) \vee \ldots \vee A_2^{p_n}(T + k)$. Therefore, $\neg X(v_p^2, T + k + 1) \in UP(T_P \cup \{A_1(T)\})$. From this, and the clause $\neg A_2(T + k + 1) \vee X(v_p^2, T + k + 1)$ we obtain $\neg A_2(T + k + 1) \in UP(T_P \cup \{A_1(T)\})$. ■

Proposition 10 Let X be a multi-valued variable of a planning problem P with domain $\mathcal{D}_X$, T_P the SATPLAN06 encoding of P, and $v_1, v_2 \in \mathcal{D}_X$ such that $\Delta_{G_X}(v_1, v_2) > k$. If A_1, A_2 are actions (including noops) such that $X(v_1) \in pre(A_1)$ and $X(v_2) \in add(A_2)$, then the set of clauses $\neg A_2(T + k - 1) \vee \neg A_1(T)$ is forward UP-redundant wrt to T_P for all valid time points.

Proof We first prove the claim for $k = 0$ with a direct proof, and then we use induction to show that the result holds for all $k > 0$.

For $k = 0$ we prove that for any pair of actions A_1, A_2 such that $X(v_1) \in pre(A_1)$ and $X(v_2) \in add(A_2)$ with $v_1, v_2 \in \mathcal{D}_X$, it holds that $\neg A_1(T) \in UP(T_P \cup \{A_2(T - 1)\})$. Let $X(v_p^2) \in pre(A_2)$ for $v_p^2 \in \mathcal{D}_X$ and assume first that $v_p^2 = v_1$. Therefore $X(v_1) \in pre(A_2)$, and by domain compatibility assumption $X(v_1) \in del(A_2)$. Note that A_2 cannot be a $noop$ because $del(A_2) \neq \emptyset$. Theory T_P contains the clause $\neg X(v_1, T) \vee A_1^{e_1}(T-1) \vee \ldots \vee A_n^{e_1}(T-1)$, where $A_c^{e_1}$, $1 \leq c \leq n$, are the actions that have $X(v_1)$ in their add effects. Since $del(A_2) \cap add(A_c^{e_1}) \neq \emptyset$, theory T_P contains the clauses $\neg A_2(T-1) \vee \neg A_c^{e_1}(T-1)$, for $1 \leq c \leq n$. Therefore, $\neg A_c^{e_1}(T-1) \in UP(T_P \cup \{A_2(T-1)\})$. From this and the clause that relates $\neg X(v_1, T)$ and $A_c^{e_1}$, we obtain $\neg X(v_1, T) \in UP(T_P \cup \{A_2(T-1)\})$ from $\neg A_1(T) \in UP(T_P \cup \{A_2(T-1)\})$ follows, since theory T_P contains the clause $\neg A_1(T) \vee X(v_1, T)$.

Now assume that $X(v_p^2) \in pre(A_2)$ and $v_p^2 \neq v_1$. Let again $A_c^{e_1}$, $1 \leq c \leq n$, be the actions that have $X(v_1)$ in their add effects. By the domain compatibility assumption, for each action $A_c^{e_1}$, $1 \leq c \leq n$ (except the noop), there is an associated precondition value $v_{p'}$ for X ($v_1 \neq v_{p'}$). For each of the above actions $A_c^{e_1}$, $1 \leq c \leq n$ say A_y such that $v_p^2 = v_{p'}$ it holds that the mutex clause $\neg A_y(T-1) \vee \neg A_2(T-1)$ is contained in theory T_P since $del(A_y) \cap pre(A_2) \neq \emptyset$, hence $\neg A_y(T-1) \in UP(T_P \cup \{A_2(T-1)\})$. For each of the above actions $A_c^{e_1}$, $1 \leq c \leq n$, say A_z, such that $v_p^2 \neq v_{p'}$, it holds by proposition 4 that the mutex clause $\neg X(v_{p'}, T-1) \vee \neg X(v_p^2, T-1)$ is in T_P. Since the clauses $\neg A_2(T-1) \vee X(v_p^2, T-1)$ and $\neg A_z(T-1) \vee X(v_{p'}, T-1)$ are also in T_P it holds for each such action $\neg A_z(T-1) \in UP(T_P \cup \{A_2(T-1)\})$. For a similar reason $\neg noop(v_1, T-1) \in UP(T_P \cup \{A_2(T-1)\})$ (since $\{\neg noop(v_1, T-1) \vee X(v_1, T-1), \neg X(v_p^2, T-1) \vee \neg X(v_1, T-1)\} \subset T_P$). From these results and the clause that relates $\neg X(v_1, T)$ and $A_c^{e_1}$, we obtain $\neg X(v_1, T) \in UP(T_P \cup \{A_2(T-1)\})$ from which $\neg A_1(T) \in UP(T_P \cup \{A_2(T-1)\})$ follows, since theory T_P contains the clause $\neg A_1(T) \vee X(v_1, T)$.

We prove now by induction on k, that if A_1, A_2 are actions such that $X(v_1) \in pre(A_1)$ and $X(v_2) \in add(A_2)$ and $\Delta_{G_X}(v_1, v_2) > k$, for $k > 0$, then $\neg A_2(T+k-1) \in UP(T_P \cup \{A_1(T)\})$.

Base case. We prove that the result holds for $k = 1$, i.e. if A_1, A_2 are actions such that $X(v_1) \in pre(A_1)$ and $X(v_2) \in add(A_2)$ and $\Delta_{G_X}(v_1, v_2) > 1$, then $\neg A_2(T) \in UP(T_P \cup$

$\{A_1(T)\})$. Let again $X(v_p^2) \in pre(A_2)$ for $v_p^2 \in \mathcal{D}_X$. From clause $\neg A_1(T) \vee X(v_1, T)$ we obtain that $X(v_1, T) \in UP(T_P \cup \{A_1(T)\})$, and from the fact mutex clauses we derive $\neg X(v_p^2, T) \in UP(T_P \cup \{A_1(T)\})$, provided that $v_p^2 \neq v_1$, which gives $\neg A_2(T) \in UP(T_P \cup \{A_1(T)\})$. Assume now that $v_p^2 = v_1$. By the definition of DTG's since there is an action (A_2) with precondition v_1 (since $v_p^2 = v_1$) and an add effect v_2, then it would hold that $\Delta_{G_X}(v_1, v_2) = 1$ contradicting the assumption $\Delta_{G_X}(v_1, v_2) > 1$, hence $v_p^2 = v_1$ cannot hold.

Inductive hypothesis. Assume that for some $k > 0$, $\neg A_2(T + k - 1) \in UP(T_P \cup \{A_1(T)\})$ holds for all pairs of actions A_1, A_2 such that $X(v_1) \in pre(A_1)$ and $X(v_2) \in add(A_2)$ and $\Delta_{G_X}(v_1, v_2) > k$.

Inductive step: We prove that $\neg A_2(T+k) \in UP(T_P \cup \{A_1(T)\})$ holds for all pairs of actions A_1, A_2 such that $X(v_1) \in pre(A_1)$ and $X(v_2) \in add(A_2)$ and $\Delta_{G_X}(v_1, v_2) > k + 1$. Let $v_p^2 \in pre(A_2)$ with $v_p^2 \in \mathcal{D}_X$, and let $A_2^{p_c}$, $1 \leq c \leq n$, be the set of actions that have $X(v_p^2)$ in their add effects. Note that $v_p^2 \neq v_1$, because otherwise $\Delta_{G_X}(v_1, v_2) = 1$, which contradicts that $\Delta_{G_X}(v_1, v_2) > k + 1$, with $k > 0$. Since $v_p^2 \neq v_1$ we have that $\Delta_{G_X}(v_1, v_p^2) > k$. From the inductive hypothesis we know that $\neg A_2^{p_c}(T + k - 1) \in UP(T_P \cup \{A_1(T)\})$ for all $1 \leq c \leq n$. Because theory T_P contains the clause $\neg X(v_p^2, T + k) \vee A_2^{p_1}(T + k - 1) \vee \ldots \vee A_2^{p_n}(T + k - 1)$, we conclude that $\neg X(v_p^2, T + k) \in UP(T_P \cup \{A_1(T)\})$. From this, and the clause $\neg A_2(T + k) \vee X(v_p^2, T + k)$ we obtain $\neg A_2(T + k) \in UP(T_P \cup \{A_1(T)\})$.

■

Proposition 11 Let X be a multi-valued variable of a planning problem P with domain $\mathcal{D}_X$, T_P the SATPLAN06 encoding of P, and $v_1, v_2 \in \mathcal{D}_X$ such that $\Delta_{G_X}(v_1, v_2) > k$. If A_1, A_2 are actions (including noops) such that $X(v_1) \in add(A_1)$ and $X(v_2) \in pre(A_2)$, then the set of clauses $\neg A_2(T + k) \vee \neg A_1(T - 1)$ is forward UP-redundant wrt to T_P for all valid time points.

Proof We prove inductively on k that $\neg A_2(T+k) \in UP(T_P \cup \{A_1(T-1)\})$ for any pair of actions A_1, A_2 such that $X(v_1) \in add(A_1)$ and $X(v_2) \in pre(A_2)$ and $\Delta_{G_X}(v_1, v_2) > k$.

Base case. For $k = 0$ we prove that for any pair of actions A_1, A_2 such that $X(v_1) \in add(A_1)$ and $X(v_2) \in pre(A_2)$ with $v_1, v_2 \in \mathcal{D}_X$, it holds that $\neg A_2(T) \in UP(T_P \cup \{A_1(T-1)\})$. By the domain compatibility assumption, there exists a fact variable $X(v_p^1)$ of G_X being a precondition of A_1. If $v_p^1 = v_2$, then it holds that $X(v_2) \in del(A_1)$, and since $X(v_2) \in pre(A_2)$ it holds that $\neg A_2(T) \in UP(T_P \cup \{A_1(T-1)\})$ due to proposition 6. Assume $v_p^1 \neq v_2$. Theory T_P contains the clause $\neg X(v_2, T) \vee A_1^{e_1}(T-1) \vee \ldots \vee A_n^{e_1}(T-1)$, where $A_c^{e_1}$, $1 \leq c \leq n$, are the actions that have $X(v_2)$ in their add effects. It holds that none of the actions $A_c^{e_1}$, $1 \leq c \leq n$ is A_1, since $X(v_1) \in add(A_1)$ and $X(v_1) \neq X(v_2)$. By the domain compatibility assumption, for each action $A_c^{e_1}$, $1 \leq c \leq n$ (except the noop), is associated a precondition value $v_{p'}$ of X ($v_2 \neq v_{p'}$). For each of the above actions $A_c^{e_1}$, $1 \leq c \leq n$ say A_y such that $v_p^1 = v_{p'}$ it holds that mutex clause $\neg A_y(T-1) \vee \neg A_1(T-1)$ is contained in theory T_P since $del(A_1) \cap pre(A_y) \neq \emptyset$, hence $\neg A_y(T-1) \in UP(T_P \cup \{A_1(T-1)\})$. For each of the above actions $A_c^{e_1}$, $1 \leq c \leq n$ say A_z such that $v_p^1 \neq v_{p'}$, it holds by proposition 4 that the mutex clause $\neg X(v_{p'}, T-1) \vee \neg X(v_p^1, T-1)$ is in T_P. Since the clauses $\neg A_1(T-1) \vee X(v_p^1, T-1)$ and $\neg A_z(T-1) \vee X(v_{p'}, T-1)$ are also in T_P, it holds for each such action $\neg A_z(T-1) \in UP(T_P \cup \{A_1(T-1)\})$. For a similar reason $\neg noop(v_2, T-1) \in UP(T_P \cup \{A_1(T-1)\})$ (since $\{\neg noop(v_2, T-1) \vee X(v_2, T-1), \neg X(v_p^1, T-1) \vee \neg X(v_2, T-1)\} \subset T_P$). From these results and the clause that relates $\neg X(v_2, T)$ and $A_c^{e_1}$, we obtain $\neg X(v_2, T) \in UP(T_P \cup \{A_1(T-1)\})$ from which $\neg A_2(T) \in UP(T_P \cup \{A_1(T-1)\})$ follows.

Inductive hypothesis. Assume that for some $k \geq 0$, $\neg A_2(T+k) \in UP(T_P \cup \{A_1(T-1)\})$ holds for all pairs of actions A_1, A_2 such that $X(v_1) \in add(A_1)$ and $X(v_2) \in pre(A_2)$ and $\Delta_{G_X}(v_1, v_2) > k$.

Inductive step: We prove that $\neg A_2(T+k+1) \in UP(T_P \cup \{A_1(T-1)\})$ holds for all pairs of actions A_1, A_2 such that $X(v_1) \in add(A_1)$ and $X(v_2) \in pre(A_2)$ and $\Delta_{G_X}(v_1, v_2) > k+1$. Let $A_2^{p_c}$, $1 \leq c \leq n$, be the set of actions that have $X(v_2)$ in their add effects, and $v^2_{p_c}$, $1 \leq c \leq n$ the associated precondition in X for each such action (by domain compatibility assumption). Clearly, $\Delta_{G_X}(v_1, v^2_{p_c}) > k$, $1 \leq c \leq n$. From the inductive hypothesis we know that $\neg A_2^{p_c}(T+k) \in UP(T_P \cup \{A_1(T-1)\})$ for all $1 \leq c \leq n$. Because theory T_P contains the clause $\neg X(v_2, T+k+1) \vee A_2^{p_1}(T+k) \vee \ldots \vee A_2^{p_n}(T+k)$, we conclude that $\neg X(v_2, T+k+1) \in UP(T_P \cup \{A_1(T)\})$. From this, and the clause $\neg A_2(T+k+1) \vee X(v_2, T+k+1)$ we obtain $\neg A_2(T+k+1) \in UP(T_P \cup \{A_1(T-1)\})$.

■

4.4.2 $londex_1$ Propagation in BB-31 Encoding

It has been shown earlier that londexes are forward UP-redundant in the SATPLAN06 encoding. Since BB-31 $>_{SUP}$ SATPLAN06, londexes are forward UP-redundant in BB-31 encoding as well. In this section we provide a counter example showing that in the BB-31 encoding, londex constraints are not backwards UP-redundant.

Consider a simple planning domain on 3 locations and a standard move action $mv(i, j)$ for moving an object from location i to location j, with $j = i+1$. Predicate p_i denote that the object is at location i. The DTG of the only variable of this domain is $G(V, E) = (\{p_1, p_2, p_3\}, \{(p_1, p_2), (p_2, p_3)\})$ and it holds that $\Delta_G(p_1, p_3) > 1$. Let T_P be the BB-31 encoding of this domain. If BB-31 is backwards UP-redundant for fact londexes then it must hold $\neg p_1(T-1) \in UP(Tp \cup \{p_3(T)\})$ for any time steps T. We show that this is not the case.

Assuming T_P is defined on a time horizon such that all clauses on time steps T and $T-1$ are complete, its clauses, for suitably defined values of i, are the following.

$\neg p_i(T) \vee mv(i-1, i, T-1) \vee Noop_i(T-1)$

$\neg mv(i-1, i, T) \vee p_{i-1}(T)$

$\neg Noop_i(T) \vee p_i(T)$

$\neg mv(i-1, i, T-1) \vee p_i(T)$

$\neg Noop_i(T-1) \vee p_i(T)$

$\neg mv(i-1, i, T-1) \vee \neg p_{i-1}(T)$

$\neg p_i(T) \vee \neg p_j(T), \quad \neg mv(i, i+1, T) \vee \neg Noop_i(T)$

The set $UP(T_P \cup \{p_3(T)\})$, for any valid point T, contains the following literals:

$\neg p_1(T), \neg p_2(T), \neg mv(1, 2, T{-}1), \neg Noop_1(T{-}1), \neg Noop_2(T{-}1), \neg mv(1, 2, T), \neg mv(2, 3, T), \neg Noop_1(T), \neg Noop_2(T)$.

It can be verified that no other inferences are possible by unit propagation, thus $\neg p_1(T-3) \notin UP(Tp \cup \{p_5(T)\})$. This leads to the following result.

Proposition 12 The londex constraints are not UP-redundant wrt the BB-31 encoding.

Similar results can be proved for action londexes. Moreover since BB-31 $>_{SUP}$ SATPLAN06 londex constraints are not backwards UP-redundant in SATPLAN06 encoding as well.

4.4.3 $londex_1$ Propagation in SATPLAN^{max}

In this section we prove that in the SATPLAN^{max} encoding all $londex_1$ constraints are UP-redundant in both directions, forward and backwards.

It was proved earlier that (clauses that correspond to) londexes ($londex_1$) are forward UP-redundant in the SATPLAN06 encoding. Since $SATPLAN^{max} >_{SUP} SATPLAN06$, londexes are forward UP-redundant in SATPLAN^{max} as well.

Proposition 13 Let X be a multi-valued variable of a planning problem P with domain $\mathcal{D}_X$, T_P the $SATPLAN^{max}$ encoding of P, and $v_1, v_2 \in \mathcal{D}_X$ such that $\Delta_{G_X}(v_1, v_2) > k$. The set of clauses $\neg X(v_2, T+k) \vee \neg X(v_1, T)$ is backward UP-redundant wrt to T_P, for all valid time points.

Proof We prove inductively on k that $\neg X(v_1, T) \in UP(T_P \cup \{X(v_2, T+k)\})$.

Base case. For $k = 0$, it follows from proposition 4 that $X(v_1)$ and $X(v_2)$ are marked mutually exclusive on all planning graph levels, therefore T_P contains the clause $\neg X(v_2, T) \vee \neg X(v_1, T)$. Hence $\neg X(v_1, T) \in UP(T_P \cup \{X(v_2, T)\})$.

Inductive hypothesis. Assume that for any pair of facts $X(v_1), X(v_2)$ and some $k \geq 0$ with $\Delta_{G_X}(v_1, v_2) > k$, it holds that $\neg X(v_1, T) \in UP(T_P \cup \{X(v_2, T+k)\})$.

Inductive step. We show that for any pair of facts $X(v_1), X(v_2)$ with $\Delta_{G_X}(v_1, v_2) > k+1$, it holds that $\neg X(v_1, T) \in UP(T_P \cup \{X(v_2, T+k+1)\})$.

By the definition of the DTG, in $G_X = (V, E)$ containing variables $X(v_1)$ and $X(v_2)$ with $\Delta_{G_X}(v_1, v_2) > k+1$, there exist (other) variables $X(v_{21}), X(v_{22}), \ldots, X(v_{2n})$ such that $\{(v_1, v_{21}), \ldots, (v_1, v_{2n})\} \subset E$, $\Delta_{G_X}(v_1, v_{2i}) = 1$ and by inductive hypothesis $\Delta_{G_X}(v_{2i}, v_2) > k$ for $X(v_{2i}) \in \{X(v_{21}), X(v_{22}), \ldots, X(v_{2n})\}$. For any variable $X(v_{2i}) \in \{X(v_{21}), X(v_{22}), \ldots, X(v_{2n})\}$, there exists the associated set of actions $\{A_1^{v_{2i}}, A_2^{v_{2i}}, \ldots, A_{m_{v_{2i}}}^{v_{2i}}\}$ ($noop(X(v_{2i})) \notin \{A_1^{v_{2i}}, A_2^{v_{2i}}, \ldots, A_{m_{v_{2i}}}^{v_{2i}}\}$) each one having $X(v_{2i})$ as an add effect, and due to domain compatibility assumption $X(v_1)$ as a precondition and delete effect. It holds that $\{\neg A_1^{v_{21}}(T) \vee X(v_{21}, T+1), \ldots, \neg A_{m_{v_{21}}}^{v_{21}}(T) \vee X(v_{21}, T+1), \ldots, \neg A_1^{v_{2n}}(T) \vee X(v_{2n}, T+1), \ldots, \neg A_{m_{v_{2n}}}^{v_{2n}}(T) \vee X(v_{2n}, T+1)\} \subset T_P$ and $\{\neg X(v_1, T) \vee X(v_1, T+1) \vee A_1^{v_{21}}(T) \vee \ldots \vee A_{m_{v_{21}}}^{v_{21}}(T) \vee \ldots \vee A_1^{v_{2n}}(T) \vee \ldots \vee A_{m_{v_{2n}}}^{v_{2n}}(T)\} \subset T_P$.

Since $\forall X(v_{2i}) \in \{X(v_{21}), X(v_{22}), \ldots, X(v_{2n})\}$ it holds that $\Delta_{G_X}(v_{2i}, v_2) > k$, by the inductive hypothesis it holds that $\neg X(v_{21}, T+1), \neg X(v_{22}, T+1), \ldots, \neg X(v_{2n}, T+1) \in UP(T_P \cup$

$\{X(v_2, T+k+1)\})$. These literals are further (unit) resolved with the binary clauses $\neg A_1^{v21}(T) \vee X(v_{21}, T+1), \ldots, \neg A_{m_{v_{21}}}^{v21}(T) \vee X(v_{21}, T+1), \ldots, \neg A_1^{v2n}(T) \vee X(v_{2n}, T+1), \ldots, \neg A_{m_{v_{2n}}}^{v2n}(T) \vee X(v_{2n}, T+1)$, giving $\{\neg A_1^{v21}(T), \ldots, \neg A_{m_{v_{2n}}}^{v2n}(T)\} \subseteq UP(T_P \cup \{X(v_2, T+k+1)\})$. The clause $\neg X(v_1, T) \vee X(v_1, T+1) \vee A_1^{v21}(T) \vee \ldots \vee A_{m_{v_{21}}}^{v21}(T) \vee \ldots \vee A_1^{v2n}(T) \vee \ldots \vee A_{m_{v_{2n}}}^{v2n}(T)$ is further resolved to the binary clause $\neg X(v_1, T) \vee X(v_1, T+1)$. But because $\Delta_{G_X}(v_1, v_2) > k$ (since $\Delta_{G_X}(v_1, v_2) > k+1$), by the inductive hypothesis it holds that $\neg X(v_1, T+1) \in UP(T_P \cup \{X(v_2, T+k+1)\})$, which further resolves the binary clause $\neg X(v_1, T) \vee X(v_1, T+1)$ to $\neg X(v_1, T)$.

■

We now prove that any of the six categories of defined action londexes is backward UP-redundant in SATPLAN^{max} encoding

Proposition 14 Let A_1, A_2 be actions and f a fact of a planning problem P such that $f \in add(A_1)$ and $f \in del(A_2)$, and T_P the $SATPLAN^{max}$ encoding of P. The set of clauses $\neg A_1(T) \vee \neg A_2(T)$, for all valid points T, is backward UP-redundant wrt the $SATPLAN^{max}$ encoding of the problem.

Proof For each valid value point T the clauses $\neg A_1(T) \vee f(T+1)$, $\neg A_2(T) \vee \neg f(T+1)$ are in T_P. Obviously $\{\neg A_2(T)\} \subseteq UP(T_P \cup \{A_1(T)\})$ and $\{\neg A_1(T)\} \subseteq UP(T_P \cup \{A_2(T)\})$. ■

Proposition 15 Let A_1, A_2 be actions and f a fact of a planning problem P such that $f \in del(A_1)$ and $f \in pre(A_2)$, and and T_P the $SATPLAN^{max}$ encoding of P. The set of clauses $\neg A_1(T-1) \vee \neg A_2(T)$, for all valid points T, is backward UP-redundant wrt the $SATPLAN^{max}$ encoding of the problem.

Proof For each valid value point T the clauses $\neg A_1(T-1) \vee \neg f(T)$, $\neg A_2(T) \vee f(T)$ are in T_P, therefore $\{\neg A_1(T-1)\} \subseteq UP(T_P \cup \{A_2(T)\})$. ■

Proposition 16 Let X be a multi-valued variable of a planning problem P with domain $\mathcal{D}_X$, T_P the $SATPLAN^{max}$ encoding of P, and $v_1, v_2 \in \mathcal{D}_X$ such that $\Delta_{G_X}(v_1, v_2) > k$, $k \geq 0$. If A_1, A_2 are actions such that $X(v_1) \in add(A_1)$ and $X(v_2) \in add(A_2)$, then the set of clauses $\neg A_2(T) \vee \neg A_1(T-k)$ is backward UP-redundant wrt to T_P for all valid time points.

Proof Since $X(v_1) \in add(A_1), X(v_2) \in add(A_2)$ for each valid value point T the clauses $\neg A_1(T) \vee X(v_1, T+1)$ and $\neg A_2(T) \vee X(v_2, T+1)$ are in T_P. Obviously $\{X(v_2, T+1)\} \subseteq UP(T_P \cup \{A_2(T)\})$, and hence by proposition 13 it holds that $\{\neg X(v_1, T+1), \neg X(v_1, T), \ldots, \neg X(v_1, T+1-k)\} \subseteq UP(T_P \cup \{A_2(T)\})$. Literals $\neg X(v_1, T+1), \neg X(v_1, T), \ldots, \neg X(v_1, T+1-k)$ are further (unit) resolved with the binary clauses $\neg A_1(T) \vee X(v_1, T+1), \neg A_1(T-1) \vee X(v_1, T), \ldots, \neg A_1(T-k) \vee X(v_1, T+1-k)$ giving $\{\neg A_1(T), \neg A_1(T-1), \ldots, \neg A_1(T-k)\} \subseteq UP(T_P \cup \{A_2(T)\})$. ■

Proposition 17 Let X be a multi-valued variable of a planning problem P with domain $\mathcal{D}_X$, T_P the $SATPLAN^{max}$ encoding of P, and $v_1, v_2 \in \mathcal{D}_X$ such that $\Delta_{G_X}(v_1, v_2) > k$, $k \geq 0$. If A_1, A_2 are actions such that $X(v_1) \in add(A_1)$ and $X(v_2) \in pre(A_2)$, then the set of clauses $\neg A_2(T) \vee \neg A_1(T-k-1)$ is backward UP-redundant wrt to T_P for all valid time points.

Proof Since $X(v_1) \in add(A_1), X(v_2) \in pre(A_2)$ for each valid value point T the clauses $\neg A_1(T) \vee X(v_1, T+1)$ and $\neg A_2(T) \vee X(v_2, T)$ are in T_P. Obviously $\{X(v_2, T)\} \subseteq UP(T_P \cup \{A_2(T)\})$, and hence by proposition 13 it holds that $\{\neg X(v_1, T), \ldots, \neg X(v_1, T-k)\} \subseteq UP(T_P \cup \{A_2(T)\})$. Literals $\neg X(v_1, T), \ldots, \neg X(v_1, T-k)$ are further (unit) resolved with the binary clauses $\neg A_1(T-1) \vee X(v_1, T)$, $\ldots, \neg A_1(T-k-1) \vee X(v_1, T-k)$ giving $\{\neg A_1(T-1), \ldots, \neg A_1(T-k-1)\} \subseteq UP(T_P \cup \{A_2(T)\})$. ■

Proposition 18 Let X be a multi-valued variable of a planning problem P with domain $\mathcal{D}_X$, T_P the $SATPLAN^{max}$ encoding of P, and $v_1, v_2 \in \mathcal{D}_X$ such that $\Delta_{G_X}(v_1, v_2) > k$, $k \geq 0$. If

A_1, A_2 are actions such that $X(v_1) \in pre(A_1)$ and $X(v_2) \in add(A_2)$, then the set of clauses $\neg A_2(T) \vee \neg A_1(T-k+1)$ is backward UP-redundant wrt to T_P for all valid time points.

Proof Since $X(v_1) \in pre(A_1), X(v_2) \in add(A_2)$ for each valid value point T the clauses $\neg A_1(T) \vee X(v_1, T)$ and $\neg A_2(T) \vee X(v_2, T+1)$ are in T_P. Obviously $\{X(v_2, T+1)\} \subseteq UP(T_P \cup \{A_2(T)\})$, and hence by proposition 13 it holds that $\{\neg X(v_1, T+1), \neg X(v_1, T), \ldots, \neg X(v_1, T+1-k)\} \subseteq UP(T_P \cup \{A_2(T)\})$. Literals $\neg X(v_1, T+1), \ldots, \neg X(v_1, T+1-k)$ are further (unit) resolved with the binary clauses $\neg A_1(T+1) \vee X(v_1, T+1), \neg A_1(T) \vee X(v_1, T), \ldots, \neg A_1(T+1-k) \vee X(v_1, T+1-k)$ giving $\{\neg A_1(T+1), \neg A_1(T), \ldots, \neg A_1(T-k+1)\} \subseteq UP(T_P \cup \{A_2(T)\})$.
■

Proposition 19 Let X be a multi-valued variable of a planning problem P with domain $\mathcal{D}_X$, T_P the $SATPLAN^{max}$ encoding of P, and $v_1, v_2 \in \mathcal{D}_X$ such that $\Delta_{G_X}(v_1, v_2) > k$, $k \geq 0$. If A_1, A_2 are actions such that $X(v_1) \in pre(A_1)$ and $X(v_2) \in pre(A_2)$, then the set of clauses $\neg A_2(T) \vee \neg A_1(T-k)$ is backward UP-redundant wrt to T_P for all valid time points.

Proof Since $X(v_1) \in pre(A_1), X(v_2) \in pre(A_2)$ for each valid value point T the clauses $\neg A_1(T) \vee X(v_1, T)$ and $\neg A_2(T) \vee X(v_2, T)$ are in T_P. Obviously $\{X(v_2, T)\} \subseteq UP(T_P \cup \{A_2(T)\})$, and hence by proposition 13 it holds that $\{\neg X(v_1, T), \ldots, \neg X(v_1, T-k)\} \subseteq UP(T_P \cup \{A_2(T)\})$. Literals $\neg X(v_1, T), \ldots, \neg X(v_1, T-k)$ are further (unit) resolved with the binary clauses $\neg A_1(T) \vee X(v_1, T), \ldots, \neg A_1(T-k) \vee X(v_1, T-k)$ giving
$\{\neg A_1(T), \neg A_1(T-1), \ldots, \neg A_1(T-k)\} \subseteq UP(T_P \cup \{A_2(T)\})$. ■

By combining the results of this and the previous section, we obtain the following property for the SATPLAN^{max} encoding.

Theorem 1 Let P be STRIPS planning domain and T_P its SATPLAN^{max} encoding. All clauses that correspond to $londex_1$ constraints derived from P are UP-redundant wrt T_P.

Note that the above result holds for `SMP` as well, as it is a simplification of SATPLAN^{max} obtained by removing UP-redundant clauses.

4.4.4 $londex_m$ Propagation in SATPLAN^{max}

By theorem 1, all $londex_1$ constraints are UP-redundant in SATPLAN^{max} (and `SMP`). In this section we provide a counter example showing that this does not hold for the stronger form of londexes $londex_m$ presented in [29].

Assume a simple planning domain where a truck transports a package between three locations p_1, p_2, p_3. Furthermore, assume three move actions $mv(p_1, p_2), mv(p_2, p_3), mv(p_3, p_1)$ for the truck, as well the action schemata $ld(p_i)$ and $unld(p_i)$, for loading and unloading respectively the package at location p_i . Predicates $t(p_i)$ and $c(p_i)$ denote that the truck and the package respectively are at location p_i, whereas $c(r)$ denotes that the package is in the truck. The corresponding noops are denoted by $noop_{p_i}$, $noop_{c_i}$, and $noop_{c_r}$.

Obviously the problem can be presented by two multi-valued variables, $TRUCK$ and $PACKAGE$. The first corresponds to the location of the truck and the second to the location of the package. Their domains are $\mathcal{D}_{TRUCK} = \{t(p_1), t(p_2), t(p_3)\}$ and $\mathcal{D}_{PACKAGE} = \{c(p_1), c(p_2), c(p_3), c(r)\}$. For example $TRUCK = t(p_1)$ expresses that the truck is at location p_1, whereas $PACKAGE = c(r)$ expresses that the package is in the truck. The DTG graphs for the two state variables are:
$G_{TRUCK}(V, E) \equiv \{\{t(p_1), t(p_2), t(p_3)\}, \{(t(p_1), t(p_2)), (t(p_2), t(p_3)), (t(p_3), t(p_1))\}\}$ and
$G_{PACKAGE}(V, E) \equiv \{\{c(p_1), c(p_2), c(p_3), c(r)\},$
$\{(c(p_1), c(r)), (c(p_2), c(r)), (c(p_3), c(r)), (c(r), c(p_1)), (c(r), c(p_2)), (c(r), c(p_3))\}\}.$

The shortest distance from $c(p_1)$ to $c(p_3)$ in $G_{PACKAGE}$ is $\Delta_{G_{PACKAGE}}(c(p_1), c(p_3)) = 2$ achievable by action $ld(p_1)$ followed by the action $unld(p_3)$, and from $t(p_1)$ to $t(p_3)$ in G_{TRUCK} is $\Delta_{G_{TRUCK}}(t(p_1), t(p_3)) = 2$ achievable by action $mv(p_1, p_2)$ followed by action $mv(p_2, p_3)$. It is easy to see that for both DTG graphs it holds that there is only one action from any transition from any value of the state variable to another. Since there is a (directed) edge $(TRUCK, PACKAGE)$ in the causal graph and $TRUCK = t(p_1), TRUCK = t(p_3)$ are prevail conditions of $ld(p_1)$ and $unld(p_3)$ respectively, the shortest distance from $c(p_1)$ to $c(p_3)$ is given by the γ function as defined in [29] (presented in section 4.2.2.2):

$$\gamma(c(p_1), c(p_3)) = min(max(\Delta_{G_{PACKAGE}}(c(p_1), c(p_3)), \Delta_{G_{TRUCK}}(t(p_1), t(p_3)) + 1)) = 3.$$

Since $\gamma(c(p_1), c(p_3)) = 3$, if $londex_m$ clauses are UP-redundant wrt SATPLAN^{max}, then in the SATPLAN^{max} encoding T_P for the above problem, it must hold that $\neg c(p_3, T + 2) \in UP(T_P \cup \{c(p_1, T)\})$ and $\neg c(p_1, T - 2) \in UP(T_P \cup \{c(p_3, T)\})$ for all time steps T.

Assuming T_P is defined on a time horizon such that all clauses on time steps T and $T - 1$ are complete, its clauses are the following.

Add axioms – clauses 5 :

$\neg t(p_1, T) \vee mv(p_3, p_1, T - 1) \vee noop_{p_1}(T - 1)$

$\neg t(p_2, T) \vee mv(p_1, p_2, T - 1) \vee noop_{p_2}(T - 1)$

$\neg t(p_3, T) \vee mv(p_2, p_3, T - 1) \vee noop_{p_3}(T - 1)$

$\neg c(r, T) \vee ld(p_1, T - 1) \vee ld(p_2, T - 1) \vee ld(p_3, T - 1) \vee noop_{c_r}(T - 1)$

$\neg c(p_i, T) \vee unld(p_i, T - 1) \vee noop_{c_i}(T - 1)$

Delete axioms – clauses 6 :

$t(p_1, T) \vee mv(p_1, p_2, T - 1) \vee \neg t(p_1, T - 1)$

$t(p_2, T) \vee mv(p_2, p_3, T - 1) \vee \neg t(p_2, T - 1)$

$t(p_3, T) \vee mv(p_3, p_1, T - 1) \vee \neg t(p_3, T - 1)$

$c(r, T) \vee \neg c(r, T - 1) \vee unld(p_1, T - 1) \vee unld(p_2, T - 1) \vee unld(p_3, T - 1)$

$c(p_i, T) \vee \neg c(p_i, T - 1) \vee ld(p_i, T - 1)$

Add effect – clauses 3 :

$\neg mv(p_1, p_2, T-1) \vee t(p_2, T), \qquad \neg noop_{p_2}(T-1) \vee t(p_2, T)$

$\neg mv(p_2, p_3, T-1) \vee t(p_3, T), \qquad \neg noop_{p_3}(T-1) \vee t(p_3, T)$

$\neg mv(p_3, p_1, T-1) \vee t(p_1, T), \qquad \neg noop_{p_1}(T-1) \vee t(p_1, T)$

$\neg ld(p_i, T-1) \vee c(r, T), \qquad \neg noop_{c_r}(T-1) \vee c(r, T)$

$\neg unld(p_i, T-1) \vee c(p_i, T), \qquad \neg noop_{c_i}(T-1) \vee c(p_i, T)$

Delete effect – clauses 4 :

$\neg mv(p_1, p_2, T-1) \vee \neg t(p_1, T)$

$\neg mv(p_2, p_3, T-1) \vee \neg t(p_2, T)$

$\neg mv(p_3, p_1, T-1) \vee \neg t(p_3, T)$

$\neg ld(p_i, T-1) \vee \neg c(p_i, T)$

$\neg unld(p_i, T-1) \vee \neg c(r, T)$

Preconditions – clauses 2 :

$\neg mv(p_1, p_2, T) \vee t(p_1, T), \qquad \neg mv(p_2, p_3, T) \vee t(p_2, T), \qquad \neg mv(p_3, p_1, T) \vee t(p_3, T)$

$\neg Noop_{p_i}(T) \vee t(p_i, T)$

$\neg ld(p_i, T) \vee c(p_i, T), \qquad \neg noop_{c_i}(T) \vee c(p_i, T)$

$\neg ld(p_i, T) \vee t(p_i, T)$

$\neg unld(p_i, T) \vee c(r, T), \qquad \neg noop_{c_r}(T) \vee c(r, T)$

$\neg unld(p_i, T) \vee t(p_i, T)$

Mutex facts – clauses 8 :

$\neg t(p_i, T) \vee \neg t(p_j, T), \forall i, j \in \{1, 2, 3\}, i \neq j$

$\neg c(p_i, T) \vee \neg c(p_j, T), \forall i, j \in \{1, 2, 3\}, i \neq j$

$\neg c(p_i, T) \vee \neg c(r, T), \forall i \in \{1, 2, 3\}$

Mutex actions – clauses 7.1 :

$\neg mv(p_1, p_2, T) \vee \neg ld(p_1, T), \qquad \neg mv(p_2, p_3, T) \vee \neg ld(p_2, T)$

$\neg mv(p_3, p_1, T) \vee \neg ld(p_3, T)$

$\neg mv(p_1, p_2, T) \vee \neg unld(p_1, T), \qquad \neg mv(p_2, p_3, T) \vee \neg unld(p_2, T)$

$\neg mv(p_3, p_1, T) \vee \neg unld(p_3, T)$

$\neg ld(p_i, T) \vee \neg noop_{p_i}(T), \qquad \neg unld(p_i, T) \vee \neg noop_{c_r}(T)$

$\neg mv(p_1, p_2, T) \vee \neg noop_{p_1}(T), \qquad \neg mv(p_2, p_3, T) \vee \neg noop_{p_2}(T)$

$\neg mv(p_3, p_1, T) \vee \neg noop_{p_3}(T)$

The literals that can be inferred by unit propagation of $c(p_1, T)$, and are therefore elements of $UP(\mathcal{T}_P \cup \{c(p_1, T)\})$ are the following (in parentheses the number that specifies the family of clauses from which the literal is derived):

$\neg c(p_2, T)(8), \neg c(p_3, T)(8), \neg c(r, T)(8),$

$\neg ld(p_2, T)(2), \neg ld(p_3, T)(2), \neg unld(p_1, T)(2), \neg unld(p_2, T)(2), \neg unld(p_3, T)(2),$

$\neg noop_{c_2}(T)(2), \neg noop_{c_3}(T)(2), \neg noop_{c_r}(T)(2),$

$\neg ld(p_1, T-1)(4), \neg ld(p_2, T-1)(3), \neg ld(p_3, T-1)(3), \neg unld(p_2, T-1)(3), \neg unld(p_3, T-1)(3), \neg noop_{c_2}(T-1)(3), \neg noop_{c_3}(T-1)(3), \neg noop_{c_r}(T-1)(3),$

$\neg c(p_2, T-1)(6), \neg c(p_3, T-1)(6), \neg c(p_2, T+1)(5), \neg c(p_3, T+1)(5),$

$\neg noop_{c_2}(T-1)(2), \neg noop_{c_3}(T-1)(2), \neg noop_{c_2}(T+1)(2), \neg noop_{c_3}(T+1)(2),$

$\neg ld(p_2, T+1)(2), \neg ld(p_3, T+1)(2), \neg unld(p_2, T-2)(3), \neg unld(p_3, T-2)(3).$

Since it holds that $\neg c(p_3, T+2) \notin UP(\mathcal{T}_P \cup \{c(p_1, T)\})$, the $londex_m$ clause $\neg c(p_1, T) \vee \neg c(p_3, T+2)$ is not forward UP-redundant in $\mathcal{T}_p$. Therefore the following result.

Proposition 20 The $londex_m$ clauses are not UP-redundant wrt the SATPLAN^{max} encoding.

Note that the above results holds for `SMP` as well, as SMP $\subset$ SATPLAN^{max}. Since SATPLAN^{max} $>_{SUP}$ BB-31 and BB-31 $>_{SUP}$ SATPLAN06, clauses $londex_m$ are not UP-redundant for BB-31 and SATPLAN06 encodings as well (the relation $>_{SUP}$ is transitive).

Domain	Problems	SP-SI	BB-SI	SMP-SI	SP-PR	BB-PR	SASE-PR	SMP-PR
`Depots`	22/22	16	16	18	17	17	17	19
`DriveLog`	20/20	16	16	16	17	17	17	17
`Zenotravel`	20/19	15	15	16	15	15	16	16
`Freecell`	20/20	4	4	6	5	5	6	6
`Satellite`	36/24	17	17	17	17	18	18	18
`Pathways`	30/30	9	9	10	12	12	16	16
`Trucks`	30/30	5	6	8	7	7	10	11
`Pipes`	50/31	17	23	23	15	24	25	27
`Storage`	30/30	15	15	15	15	15	15	16
`TPP`	30/30	27	28	28	28	29	30	30
`Elevators`	30/30	9	9	12	12	13	14	14
`ScanAnalyser`	30/23	17	17	19	15	16	18	18
`Sokoban`	30/30	2	2	4	2	5	2	7
`Transport`	30/21	11	11	12	11	11	13	13
Total	408/360	180	188	204	188	204	217	228

Table 1: Number of problems solved by each encoding in different domains.

4.5 Experimental evaluation

In this section we present the results of the experimental comparison of various encodings discussed earlier, in domains from planning competitions. Our implementation is an extension of the `SATPLAN06` system with new encodings for `BLACKBOX` and `SMP` as well as the integration of `precosat`. Hence, all experiments are runs of the same system with different values for the parameters *encoding* and *solver*. The experiments were run on an IBM X3650 with Intel Xeon processors at 2.0 GHz and 32GB of RAM, running under CentOS 5.2.

Table 1 presents the number of problems solved with different combinations of encodings and SAT solvers, within a CPU time limit of 2500 seconds. The encodings compared are `SATPLAN06` (encoding SATPLAN06-4), `BLACKBOX` (encoding BB-31), and `SMP`. We also conducted experiments of `SASE` planner with the same time limit. The SAT solvers that are used are `siege` [109] and `precosat` [16] version 236, a newer system that seems to outperform Siege and many other solvers that we have tested on a large number of planning domains. In Table 1 (as well as Table

Domain-Problem	SMP	BB	SP
Depots-11	176	1674	2134
DriveLog-16	897	1156	2453
Zenotravel-15	84	307	383
Pathways-17	971	980	1940
Trucks-8	161	637	1140
TPP-21	1580	1908	2554
Pipes-12	189	348	1429
Transport-4	81	312	563
Sokoban-13	474	1869	-
Elevator-21	2099	2424	-
ScanAnalyser-8	59	208	-

Table 2: Run times in seconds for different encodings of problems. A dash indicates CPU timeout.

2) `SATPLAN06` is denoted by SP, `BLACKBOX` by BB, whereas `siege` by SI and `precosat` by PR.

The entries under ”Problems” in Table 1 are of the form p/q, where p is the total number of problems contained in each domain, and q the number of problems for which either one of the methods found a solution, or they all reached their CPU limit and terminated without a solution. Hence, $p-q$ is the number of problems that were not solved by any of the systems due to memory problems at parsing or solving time.

Table 2 presents characteristic run times of different encodings on some of the hardest problems that have been solved by both `BLACKBOX` and `SMP`. All times were obtained with `precosat` as the underlying SAT solver, and a CPU time limit of 3600 seconds.

The relative performance of the different encodings, as depicted in Tables 1 and 2, is consistent with the theoretical results obtained in earlier sections. Indeed, `BLACKBOX` outperforms `SATPLAN06`, whereas `SMP` dominates all other encodings. Moreover, solution times improve when `precosat` instead of `siege` is used as the SAT solver.

4.6 Binary Constraints in Planning as Satisfiability

In the spirit of this work, a natural question arises: Can we identify new families of binary clauses that are useful in achieving more propagation, and therefore speeding-up the solving in the planning as SAT framework? As the vast majority of SAT-solvers use unit propagation as their constraint propagation method, we are interested only in *non UP-redundant* binary clauses.

There are two approaches to answer this question:

I. Identify analytically new kinds of binary clauses that have a intuitive meaning for planning and are not *UP-redundant*.

II. Discover implied clauses automatically, with the use of suitable off-line tool.

In the following we explore both options. Regarding the automated clause discovery, the goal is not to develop a preprocessing tool to be used in plan generation, but rather an off-line explorative system that may reveal new useful binary clauses. In other words, the question to be answered here is whether useful constraints exist, rather than how they can be found efficiently. Clearly, if such constraints do not exist, there is no motive in devising efficient algorithms for computing them.

4.6.1 Prevail constraints

According to the state-variable representation (see Chapter 3), a prevail condition of an action is a value of a state variable that is required by the action in ordered to be applicable, but is not changed by that action. In STRIPS representation, a prevail condition is defined as follows:

Definition 35 A fact f is a prevail condition of an action $A =< pre(A), add(A), del(A) >$ if $f \in pre(A) \wedge f \notin del(A)$.

If an action is applied at a time step, all its prevail conditions must be true at the next step. The intuition for this is that since action A is applied at time T, then its prevail condition f must also be true at time step T (since $f \in pre(A)$). Since f is true at time step T, in order for f to be false at time $T+1$, an action A' such that $f \in del(A')$ must be applied at time step T (in order to delete the fact f). But any such action A' interferes with action A (since $f \in (del(A') \cap pre(A))$) and since A is true at time step T, A' is false at time step T. We prove this intuition formally in next proposition.

Proposition 21 Let A be an action of a planning problem P, and f a prevail condition of A. Then for all the valid time steps T the binary clause $\neg A(T) \vee f(T+1)$ is an implied clause of the SATPLAN^{max} encoding of P.

Proof Let $A_1, \ldots, A_n$ be all the actions of the problem P deleting f ($f \in del(A_i)$ for all $1 \leq i \leq n$), and $\mathcal{T}_p$ the SATPLAN^{max} translation of P. It holds for the set of clauses $\mathbb{C} = \{\neg A(T) \vee f(T), f(T+1) \vee A_1(T) \vee \ldots \vee A_n(T) \vee \neg f(T), \neg A(T) \vee \neg A_1(T), \ldots, \neg A(T) \vee \neg A_n(T)\}$ that $\mathbb{C} \subset \mathcal{T}_p$. It is easy to verify that $f(T+1) \in UP(\mathbb{C} \cup \{A(T)\})$ hence $f(T+1)) \in UP(T_p \cup \{A(T)\})$ since $\mathbb{C} \subset \mathcal{T}_p$, therefore $\neg A(T) \vee f(T+1)$ is an implied constraint of $\mathcal{T}_p$, which is also forward UP-redundant.

■

We prove next that the implied clause $\neg A(T) \vee f(T+1)$ is not backward UP-redundant wrt the SATPLAN^{max} encoding.

Proposition 22 Let A be an action of a planning problem P, f a prevail condition of A and $\mathcal{T}_p$ the SATPLAN^{max} translation of P. Then, for all the valid time steps T the implied binary clause $\neg A(T) \vee f(T+1)$ is *not* backward UP-redundant wrt the theory $\mathcal{T}_p$.

Proof Assume a planning problem P with two persons P_1 and P_2 and three rooms R_1, R_2 and R_3 with a light switch only in R_1. Any of the persons can switch on the light provided she is in room R_1. Only P_1 can move, and the only valid moves are from R_1 to any of the other two rooms. Initially the switch is off and the two persons are in room R_1. There are four actions: The actions $TuOn(P_1)$ and $TuOn(P_2)$ with semantics that the person P_1 or P_2 turns the switch on respectively. The other two actions are $MvR(P_1, R_2)$ and $MvR(P_1, R_3)$ with semantics that the person P_1 moves to room R_2 or to room R_3 respectively. The goal is to turn the switch on. Obviously there are two optimal plans of a planning horizon one with just one action, any of the persons to turn on the switch. The problem (STRIPS) description is $P =< I, G, A >$ where:

$I = \{inroom(P_1, R_1), inroom(P_2, R_1), switchoff\}$

$G = \{switchon\}$ and

$A = \{TuOn(P_1), TuOn(P_2), MvR(P_1, R_2), MvR(P_1, R_3)\}$

where the action descriptions are:

$pre(TuOn(P_1)) = \{inroom(P_1, R_1), switchoff\}, add(TuOn(P_1)) = \{switchon\}$

and $del(TuOn(P_1)) = \{switchoff\}$

$pre(TuOn(P_2)) = \{inroom(P_2, R_1), switchoff\}, add(TuOn(P_2)) = \{switchon\}$

and $del(TuOn(P_2)) = \{switchoff\}$

$pre(MvR(P_1, R_2)) = \{inroom(P_1, R_1)\}, add(MvR(P_1, R_2)) = \{inroom(P_1, R_2)\}$

and $del(MvR(P_1, R_2)) = \{inroom(P_1, R_1)\}$

$pre(MvR(P_1, R_3)) = \{inroom(P_1, R_1)\}, add(MvR(P_1, R_3)) = \{inroom(P_1, R_3)\}$

and $del(MvR(P_1, R_3)) = \{inroom(P_1, R_1)\}$

From the action descriptions above it holds that the fact $inroom(P_1, R_1)$ is a prevail condition of action $TuOn(P_1)$, since $inroom(P_1, R_1) \in pre(TuOn(P_1))$ and $inroom(P_1, R_1) \notin del(TuOn(P_1))$.

The clauses of the SATPLAN^{max} translation $\mathcal{T}_p$ of the above problem P for horizon one are (where the last argument of fact/action denotes the time step):

Initial conditions – clauses 1 :

$inroom(P_1, R_1, 0), \quad inroom(P_2, R_1, 0), \quad switchoff(0)$

Add axioms – clauses 5 :

$\neg switchon(1) \vee TuOn(P_1, 0) \vee TuOn(P_2, 0)$

$\neg inroom(P_1, R_1, 1) \vee noop_{inroom(P_1,R_1)}(0)$

$\neg inroom(P_2, R_1, 1) \vee noop_{inroom(P_2,R_1)}(0)$

$\neg inroom(P_1, R_2, 1) \vee MvR(P_1, R_2, 0)$

$\neg inroom(P_1, R_3, 1) \vee MvR(P_1, R_3, 0)$

$\neg switchoff(1) \vee noop_{switchoff}(0)$

Delete axioms – clauses 6 :

$inroom(P_1, R_1, 1) \vee MvR(P_1, R_2, 0) \vee MvR(P_1, R_3, 0) \vee \neg inroom(P_1, R_1, 0)$

$inroom(P_2, R_1, 1) \vee \neg inroom(P_2, R_1, 0)$

$switchoff(1) \vee TuOn(P_1, 0) \vee TuOn(P_2, 0) \vee \neg switchoff(0)$

Add effect – clauses 3 :

$\neg TuOn(P_1, 0) \vee switchon(1), \quad \neg TuOn(P_2, 0) \vee switchon(1)$

$\neg MvR(P_1, R_2, 0) \vee inroom(P_1, R_2, 1), \quad \neg MvR(P_1, R_3, 0) \vee inroom(P_1, R_3, 1)$

$\neg noop_{inroom(P_1,R_1)}(0) \vee inroom(P_1, R_1, 1)$

$\neg noop_{inroom(P_2,R_1)}(0) \vee inroom(P_2, R_1, 1)$

$\neg noop_{switchoff}(0) \vee switchoff(1)$

Delete effect – clauses 4 :

$\neg TuOn(P_1, 0) \vee \neg switchoff(1)$

$\neg TuOn(P_2, 0) \vee \neg switchoff(1)$

$\neg MvR(P_1, R_2, 0) \vee \neg inroom(P_1, R_1, 1)$

$\neg MvR(P_1, R_3, 0) \vee \neg inroom(P_1, R_1, 1)$

Preconditions – clauses 2 :

$\neg TuOn(P_1, 0) \vee inroom(P_1, R_1, 0), \quad \neg TuOn(P_2, 0) \vee inroom(P_2, R_1, 0)$

$\neg TuOn(P_1, 0) \vee switchoff(0), \quad \neg TuOn(P_2, 0) \vee switchoff(0)$

$\neg MvR(P_1, R_2, 0) \vee inroom(P_1, R_1, 0)$

$\neg MvR(P_1, R_3, 0) \vee inroom(P_1, R_1, 0)$

$\neg noop_{inroom(P_1,R_1)}(0) \vee inroom(P_1, R_1, 0)$

$\neg noop_{inroom(P_2,R_1)}(0) \vee inroom(P_2, R_1, 0)$

$\neg noop_{switchoff}(0) \vee switchoff(0)$

Mutex facts – clauses 8 :

$\neg switchon(1) \vee \neg switchoff(1)$

$\neg inroom(P_1, R_1, 1) \vee \neg inroom(P_1, R_2, 1)$

$\neg inroom(P_1, R_1, 1) \vee \neg inroom(P_1, R_3, 1)$

$\neg inroom(P_1, R_2, 1) \vee \neg inroom(P_1, R_3, 1)$

Mutex actions – clauses 7.1 :

$\neg TuOn(P_1, 0) \vee \neg TuOn(P_2, 0)$

$\neg TuOn(P_1, 0) \vee \neg MvR(P_1, R_2, 0)$

$\neg TuOn(P_1, 0) \vee \neg MvR(P_1, R_3, 0)$

$\neg TuOn(P_1, 0) \vee \neg noop_{switchoff}(0)$

$\neg TuOn(P_2, 0) \vee \neg noop_{switchoff}(0)$

$\neg MvR(P_1, R_2, 0) \vee \neg noop_{inroom(P_1,R_1)}(0)$

$\neg MvR(P_1, R_3, 0) \vee \neg noop_{inroom(P_1,R_1)}(0)$

Goals – clauses 1 :

$switchon(1)$

It is easy to verify that

$UP(\mathcal{T}_p \cup \{\neg inroom(P_1, R_1, 1)\}) =$

$\{inroom(P_1, R_1, 0), inroom(P_2, R_1, 0), switchoff(0), switchon(1),$

$inroom(P_2, R_1, 1), \neg switchoff(1), \neg noop_{switchoff}(0),$

$noop_{inroom(P_2,R_1)}(0), \neg inroom(P_1, R_1, 1), \neg noop_{inroom(P_1,R_1)}(0)\},$

hence $\neg TuOn(P_1, 0) \notin UP(\mathcal{T}_p \cup \{\neg inroom(P_1, R_1, 1)\})$. From this it follows that the implied clause $\neg TuOn(P_1, 0) \vee inroom(P_1, R_1, 1)$ is not backward UP-redundant in the theory $\mathcal{T}_p$.

■

Proposition 23 Let A be an action of a planning problem P, f a prevail condition of A, and $\mathcal{T}_p$ the SATPLAN^{max} translation of P. The implied binary clause $\neg A(T) \vee f(T+1)$ is not UP-redundant wrt $\mathcal{T}_p$ and valid time step T.

Proof Follows directly from proposition 22.

■

Note that the above results holds for `SMP` as well since `SMP` $>_C$ `SATPLAN`^{max}. Since `SATPLAN`^{max} $>_{SUP}$ `BB-31` and `BB-31` $>_{SUP}$ `SATPLAN06`, propositions 22 and 23 hold for `BB-31` and `SATPLAN06` encodings as well (the relation $>_{SUP}$ is transitive).

Proposition 23 implies that prevail constraints can be, in principle, useful when added to the SATPLAN^{max} translation of a problem since they can achieve more propagation for a SAT solver that uses unit propagation as its constraint propagation method. Another interesting side effect when adding these constraints, is that many action mutexes can become UP-redundant and hence can be removed from the theory. When a prevail constraint $\neg A(T) \vee f(T+1)$ is added to the encoding, if there is an action $A'(T)$ such that $A'(T)$ is mutex in the theory with $A(T)$, then the action mutex constraint $\neg A(T) \vee \neg A'(T)$ can be safely removed (because it's UP-redundant) if A' deletes f or if it adds an f' which is mutex with f at layer $T + 1$. For example in the above example if we add the prevail constraint $\neg TuOn(P_1, 0) \vee inroom(P_1, R_, 1)$ then the action mutexes clauses $\neg TuOn(P_1, 0) \vee \neg MvR(P_1, R_2, 0)$ and $\neg TuOn(P_1, 0) \vee \neg MvR(P_1, R_3, 0)$ can be safely removed as UP-redundant.

We implemented a version of the `SMP` encoding that at all layers all the prevail constraints for all the (no NOOPs) actions are added, and any action mutexes that become UP-redundant are removed. This new encoding can be selected by the user by providing a suitable value for the parameter *encoding* of the system. Our experiments showed better planning times of about 10-20% for the problems of all domains, except *Trucks* where we get a speed-up of about 50%, and all the action mutexes are removed as UP-redundant.

4.6.2 Automatically computed binary constraints

Binary constraints may speed up SAT-based planning provided that they enhance the constraint propagation in the underlying solver. For instance, in the `SMP` encoding there are at least two type of meaningful clauses that we formally proved that are not UP-redundant: the *prevail* constraints (section 4.6.1) presented in this work, and the $londex_m$ constraints of [29] discussed in 4.2.2.2. We implemented *ImpBinPlan*, a tool that finds automatically non UP-redundant binary constraints. *ImpBinPlan* must be seen as an exploratory tool for the discovery of new families of clauses, rather as a preprocessing method that targets directly to reduce plan times. In fact, as it will come apparent later, *ImpBinPlan* is a very slow tool.

Our aim is to find 'general' clauses: for example, $londex_m$ and *prevail* clauses are general as they can be found in many domains. Moreover, they should not be depended on the initial state or goals of a specific problem, but merely on the structure of the domain. As with SMP, *ImpBinPlan* takes as input a domain and a problem description. For this reason, all goals are removed from the SMP encoding of the problem that is analyzed. On the other hand, as *ImpBinPlan* builds on tools developed within the SATPLAN framework, there is no easy way to remove the initial conditions. For this reason we follow an 'indirect' approach: Suppose we want to find binary constraints in a domain for a planning horizon k. We build a planning graph of $l + k$ layers, where l is the first layer where the graph plan levels off. Then we take the sub-graph from layer l to $l+k$ and translate it to SMP encoding (without any goals). There is another reason for this approach. Since theory T_P, derived as explained, contains only leveled-off layers, any clause at each layer is an exact copy of one in the previous layers. For example if there is a mutex fact clause $\neg f_1(l) \vee \neg f_2(l)$ at layer l then $\neg f_1(l + i) \vee \neg f_2(l + i), \forall i \in [1, \ldots, k] \in T_P$. It is not hard to prove that due to this 'symmetry' of the clauses over layers, if an implied clause $v_1(x) \vee v_2(y)$ is found, then is safe to conclude that $v_1(x + d) \vee v_2(y + d)$ is also an implied clause for $d > 0$.

ImpBinPlan takes as input a SMP theory T_P, and a number of (user-defined) parameter values. Assume that the first layer of T_P is l and the last is $l + k$. At a high level, the algorithm iteratively selects all pairs of variables x_1, x_2 ($x_1 \neq x_2$). For each pair x_1, x_2 and a user-defined layer distance $dist$ it tests for each of the four clauses (for the four possible polarities) $c_1 = x_1(l) \vee x_2(l+dist)$, $c_2 = x_1(l) \vee \neg x_2(l+dist)$, $c_3 = \neg x_1(l) \vee x_2(l+dist)$ and $c_4 = \neg x_1(l) \vee \neg x_2(l+dist)$ if they are implied clauses. For any clause $c_i, i \in \{1, 2, 3, 4\}$ found to be an implied clause, $dist$ is increased by one, and the process repeats until a non-implied clause is found or $dist = k$. The user can choose to omit from implication testing any of the four polarities. Moreover, the user can select any subset of the set of four combinations that can be generated depending on whether the literals x_1, x_2 of the implied clause corresponds to a fact or an action. This is sensible since facts are substantially fewer than action in most domains, and actions are usually tightly coupled with facts through propagation. Any implied clauses found are stored in an queue in the order they are found.

The algorithm tests if a clause is implied by adding to the theory the unit clauses for the opposite polarity, and then running a SAT-solver (we used `precosat` [16] version 236). For testing the four clauses c_1, c_2, c_3, c_4 of the previous paragraph, the corresponding theories $T_P^1 = T_P \cup \{\neg x_1(l), \neg x_2(l+dist)\}, T_P^2 = T_P \cup \{\neg x_1(l), x_2(l+dist)\}, T_P^3 = T_P \cup \{x_1(l), \neg x_2(l+dist)\}$ and $T_P^4 = T_P \cup \{x_1(l), x_2(l+dist)\}$ are generated. Obviously c_i is implied if and only if T_p^i is inconsistent for $i \in \{1, 2, 3, 4\}$. The user can impose a cut-off time limit to the SAT solver for each test performed in a pair of variables and distance. Every time the solver times out, the corresponding clause is considered non-implied by the algorithm, and the algorithm proceeds to the next pair of variables.

In the final step, *ImpBinPlan* reads the queue with the implied clauses (in FIFO order). For the read clause c, if it is not UP-redundant wrt T_P, T_P is updated to $T_P \cup \{c\}$. This is done to avoid the insertion (or discovery) of implied clauses that are not UP-redundant and hence not useful. We note that the set of implied clauses found (the non-UP redundant) is not neccesarily a minimal one with respect to the implied clauses found by *ImpBinPlan* in the first step. The UP-redundancy test for a clause c is straightforward. If $c = \neg x_1(l) \vee \neg x_2(l')$ then c is UP-redundant wrt T_P if $\neg x_1(l) \in UP(T_P \cup \{x_2(l')\}) \bigwedge \neg x_2(l') \in UP(T_P \cup \{x_1(l)\})$

4.6.3 Experimental findings

We conducted a number of experiments with the *ImpBinPlan* constraint discovery tool in different planning domains with the following two objectives

- Identify other forms of binary clauses that may be present in planning domains and are not UP-redundant.

Domain	F/F/0	F/A/0	F/F/-/-	F/F/+/-	F/F/+/+	F/A/-/-
`Freecell`	N	N	Y	N	N	Y
`Path`	N	N	Y	N	N	Y
`Barman`	N	Y	Y	N	N	Y
`Trucks`	N	N	Y	N	N	Y
`Visit`	N	N	N	N	N	N
`Sokobahn`	N	N	Y	N	N	Y
`Transport`	N	N	Y	N	N	Y
`Openstacks`	N	N	Y	N	N	Y
`Pipes`	N	N	Y	N	N	Y
`Storage`	N	N	Y	N	N	Y

Table 3: Types of binary clauses searched in various domains. F/F and F/A denote fact,fact and fact,action pairs respectively. The postfix 0 denotes pairs of literals that refer to the same time step. The postfixes -/-, etc. refer to the polarity of the clauses. An entry Y means that the domain contains clauses of the corresponding type, and N means the opposite.

- Determine experimentally whether these constraints improve SAT solving.

The type of binary clauses sought are shown in Table 3. All clauses are of the form $\neg p(T) \vee \neg q(T+k)$, where p and q are literals, and $k \geq 0$. Therefore, the discovered clauses are a generalization of the londexes of [29], that are restricted to negative fact literals. The exact value of k is domain dependent, and explained below.

The F/F columns correspond to binary clauses of the form $f_1 \vee f_2$ where f_1 and f_2 are literals that correspond to facts, whereas F/A denotes clauses of the form $f \vee a$ where f and a are fact and action literals respectively. The +/- notation refers to the polarity of the literals in the clause, e.g. F/F/+/- corresponds to clauses of the form $f_1 \vee \neg f_2$ where f_1 and f_2 are fact atoms. For the fact,action pairs *ImpBinPlan* was only run for clauses where both of its literals are negative. This restriction is due to computational considerations that are explained later. The second and third column in Table 3, i.e. F/F/0 and F/A/0 correspond to the binary clauses on literals that refer to the same time point. For F/F/0, all four different literal polarities have been tried. For F/A/0 only clauses with literals with opposite polarities were searched. Finally, an entry Y in the table means that the corresponding type of binary clauses is present in the domain, and N the opposite.

The findings presented in Table 3, are based on runs with relatively small problems from the domains that are considered. This is a restriction imposed by the complexity of the problem. Indeed, there are domains that, even for small problems, induce hundreds of thousands or millions of clauses. To answer the question of whether each such clauses is implied, a SAT problem needs to be solved. As a result, in some domains, finding all binary constraints requires running *ImpBinPlan* for several weeks.

The first conclusion that follows from the experiments is that no new implied binary clauses on fact variables were found, except for those that are on two negative fact literals. On the other hand, almost all domains (except for `Visit` that no additional clause was found) have londex constraints that are on negative fact literals. Probably the most interesting finding is the existence of binary constraints on one fact and one action literal. Interestingly, these binary clauses appear in almost all domains. The question that needs to be answered next is whether these new constraints can improve the search for a plan.

4.6.4 Utility of the extra constraints

In order to determine the effects of the discovered binary clauses on the performance on `SMP`, we conducted experiments on various domains. The constraints found by *ImpBinPlan* were added to `SMP`, along with the standard clauses of the encoding. Then, the run times of `SMP` with and without the extra binary clauses are compared.

The results are presented in Table 4, and refers to SAT instances derived from planning problems with a fixed planning horizon. The column under ”# F/A clauses” lists the number of binary clauses of the form $\neg f(T) \vee \neg A(T + dist)$ found by *ImpBinPlan*. Note that f is a fact atom and

Problem	$dist$	# F/A clauses	Time W/O	Time W
`Barman-Prob8-Hor49`	25	1943092	604	756
`Barman-Prob9-Hor49`	25	2179572	988	1240
`Sokoban-Prob8-Hor119`	16	1178280	655	601
`Sokoban-Prob9-Hor195`	16	2062693	292	288
`Storage-Prob16-Hor10`	8	287640	208	228
`Storage-Prob17-Hor11`	8	472837	1394	3680
`Openstacks-Prob7-Hor30`	11	62353	838	1108
`Openstacks-Prob8-Hor30`	7	159289	1259	1484
`Pathways-Prob9-Hor18`	12	60256	632	505

Table 4: Impact of the binary constraint discovered by *ImpBinPlan* on the performance of `SMP`. The first column refers to the domain, where the numbers X and Y in Prob$X$-Hor$Y$ designate the problem number and the planning horizon respectively. The numbers in column "$dist$" are the values for parameter $dist$, explained in section 4.6.2, whereas under "# F/A clauses" is the number of binary constraints found in each domain. Finally, the last two columns provide the run times in seconds for `SMP` without ("Time W/O") and with ("Time W") the constraints found by *ImpBinPlan*.

A and action atom, whereas the value for parameter $dist$ that has been used in each run is listed in the second column. The experimental results seem to imply that the new family of constraints that were found fail to improve significantly the performance of the underlying solver. In fact, in some cases they degrade the run time considerably.

4.7 Conclusions

In this chapter we compared different encodings of planning as satisfiability wrt the propagation they achieve in a modern SAT solver. Our theoretical results explain some of the differences observed in the performance of various planners. One interesting finding is that `BLACKBOX` encoding is stronger than the one of `SATPLAN06`. Thus, new encodings of planning as satisfiability need to be compared with both systems. Another practical outcome of our results is `SMP`, a new encoding that renders londex constraint (on a single DTG) redundant, and seems to offer performance improvements in a number of domains. Finally, we investigate the effects of adding more implied binary constraints to the SAT encoding, and showed experimentally that this does not bring substantial gains.

Chapter 5

Propositional Planning as Optimization

The PSP planning system

Planning as Satisfiability, that was presented in the previous chapters, is an important approach to optimal propositional planning. Although optimality is highly desirable, for large problems it comes at a high, often prohibitive, computational cost.

This chapter extends planning as propositional satisfiability to planning as pseudo-boolean optimization. The approach has been implemented in a planner called PseudoSATPLAN, that follows the classic solve and expand method of the SATPLAN algorithm, but at each step it seeks to maximize the number of goals that can be achieved. The utilization of the achieved goals at subsequent steps opens up the possibility of implementing various strategies. The method essentially splits a planning problem into smaller subproblems, and employs various techniques for solving them fast. Although PseudoSATPLAN cannot guarantee the optimality of the generated plans, it aims at computing solutions of good quality. Experimental results show that PseudoSATPLAN can generate parallel plans of high quality for problems that are beyond the reach of the existing implementations of optimal planning as satisfiability framework.

5.1 Introduction

The SATPLAN approach [75], is a successful approach to optimal STRIPS planning. Although it generates (parallel) plans with optimal makespan, optimality comes at a high computational cost. The question that naturally arises is whether it is possible to maintain the general idea of formulating planning as a boolean constraint satisfaction problem, but at the same time generate suboptimal plans, trading optimality for efficiency and scalability.

This work answers the above question in the affirmative. It presents a new planning system, called PseudoSATPLAN, or PSP for short, that follows the classic solve and expand approach and works in two parts. During the first part, called *optimization part*, PSP seeks to maximize the number of goals that can be achieved within a fixed planning horizon. If this number is smaller than some user supplied value, the planning horizon is extended and the procedure iterates. As soon as a plan that attains the specified number of goals is found, PSP enters its, second, *satisfaction part*. The actions of the plan that is returned as the result of the optimization part are used to identify a new initial state and, consequently, a new planning problem, that is solved by a simple call to the classic SATPLAN algorithm. The concatenation of the two plans is a solution to the original problem.

The goals that are attained during the optimization part are added to the problem as intermediate facts, that must be established in the plan that is being generated. In terms of the underlying propositional satisfiability problem that needs to be solved, these facts are unit clauses. The propagation that they trigger yields a smaller, and usually easier, SAT instance. The optimization part is divided in three *optimization phases*. These phases differ mainly in the degree of use of these intermediate goals; as the planning horizon increases and the associated optimization problem becomes more difficult, the number of intermediate goals that are added to the theory increases as well.

There are several possibilities regarding the treatment of these intermediate goals, each yielding a different plan search strategy. When `PSP` applies the *fixed goals* strategy, it searches for plans in which any intermediate goal that has been achieved in previously computed solutions, is required to be true at the time step it was first established. In contrast, the *sliding goals* strategy, moves the intermediate goals towards later time points, at a pace that can be defined by the user. Finally, in order to generate plans of good quality, `PseudoSATPLAN` employs a technique called *back stepping*, as well as a *plan improvement* mechanism.

Experimentation in twelve domains from various planning competitions shows that the method can solve 65 more problems than `SMP`, presented in the previous chapter, one of the most efficient implementations of the planning as satisfiability approach. To a great extent the success of the method can be attributed to a strong SAT model of planning such as `SMP`, as well as the availability of powerful SAT solvers such as `precosat` [16].

`PseudoSATPLAN` is directly applicable to planning for conjunctive goals. The great majority of the publicly available domains and problems, including those of the Planning Competitions, belong to this class. Nevertheless, there are ways to extend the applicability of the techniques employed by `PSP` to a broader range of problems. As this work is a first investigation of the planning as optimization framework, these more advanced techniques fall outside its scope.

5.2 Pseudo-boolean optimization and planning

This section introduces the main tools and techniques that are used in `PseudoSATPLAN`. Our analysis refers exclusively to STRIPS planning.

As we did in Chapter 4, we assume STRIPS planning problems $P =< I, G, A >$ as presented in Chapter 3. Recall that each action $a \in A$ has preconditions $pre(a)$, add effects $add(a)$, and delete effects $del(a)$. Recall that a *(Linear) Pseudo-boolean constraint* (PB-constraint) is an inequality of the form $\sum a_i x_i \geq b$, where x_i is a boolean variable, and a_i, b integers as presented in Chapter 2. A PB-constraint is the generalization of a clause. Indeed, a propositional clause $x_1 \vee \ldots \vee x_m \vee \neg x_{m+1} \vee \ldots \neg \vee x_n$, where each x_i is an atom, translates into the PB-constraint $x_1 + \ldots + x_m - x_{m+1} - \ldots - x_n \geq m - n + 1$. Following this translation, the clauses of the SMP encoding presented in Chapter 4 translate to PB formulas as shown below:

1. Unit clauses for the initial state.

   Translates to: $f(0) \geq 1$ in PB.

2. $A(T) \rightarrow f(T)$, for every action A and fact f s.t. $f \in pre(A)$.

   Translates to: $-A(T) + f(T) \geq 0$ in PB.

3. $A(T) \rightarrow f(T+1)$, for every action A and fact f s.t. $f \in add(A)$.

   Translates to: $-A(T) + f(T+1) \geq 0$ in PB.

4. $A(T) \rightarrow \neg f(T+1)$, for every action A and fact f s.t. $f \in del(A)$.

   Translates to: $-A(T) - f(T+1) \geq -1$ in PB.

5. $f(T) \rightarrow A_1(T-1) \vee \ldots \vee A_m(T-1)$, for every fact f and all actions A_i, $1 \leq i \leq m$ (including the noops) s.t. $f \in add(A_i)$.

   Translates to: $-f(T) + A_1(T-1) + \ldots + A_m(T-1) \geq 0$ in PB.

6. $\neg f(T) \rightarrow A_1(T-1) \vee \ldots \vee A_m(T-1) \vee \neg f(T-1)$, for every fact f and all actions A_i, $1 \leq i \leq m$ s.t. $f \in del(A_i)$.

   Translates to: $+f(T) - f(T-1) + A_1(T-1) + \ldots + A_m(T-1) \geq 0$ in PB.

7.1 $\neg A_1(T) \vee \neg A_2(T)$, for every pair of mutex actions A_1, A_2 such that the set $del(A_1) \cap pre(A_2)$ is non-empty.

Translates to: $-A_1(T) - A_2(T) \geq -1$ in PB.

8 $\neg f_1(T) \vee \neg f_2(T)$, for every pair of facts f_1, f_2 that are mutex at time T.

Translates to: $-f_1(T) - f_2(T) \geq -1$ in PB.

PseudoSATPLAN generates the PB formulation of a planning problem by translating its SMP model into a set of inequalities, following the direct method described above.

Note that we deliberately *do not* translate the goals of the SMP model into inequalities $\forall g \in G$, $g(T) \geq 1$, where T is the planning horizon since we do not want to commit to *all* the goals to be true in the final state. Instead we want to *maximize* the number of goals that can be satisfied. We do that by defining the appropriately objective function f^G:

For a problem with a set of goals $G = \{g_1, \ldots, g_k\}$, the objective function f^G is defined on the set of temporally extended goals as

$$max : f^G(g_1(T), \ldots, g_k(T)) = \sum_{g_i \in G} g_i(T)$$

where T is the planning horizon. This is the underlying PBO model employed by the algorithms described in the following sections.

The performance of minisat+, as well as a few other PB optimizers that were tested, on problems arising from planning are rather unsatisfactory. Therefore, PseudoSATPLAN employs minisat+ only as a system that translates PB constraints into CNF using the BDD trees method that was briefly presented in chapter 2 section 2.3, and the resulting problems are solved by precosat. More details are given in the next section.

5.3 The PseudoSATPLAN Algorithm

The PSP-plan algorithm presented in Algorithm 1 is the main planning procedure of PseudoSATPLAN. It takes as input a planning problem $P =< I, G, A >$, a number of parameters (the most important listed in the Require statement of the algorithm) that are discussed below, and returns a $plan$. PSP operates in two parts, the *optimization* and the *satisfaction* part.

5.3.1 Optimization part

In the optimization part, PSP (lines 1-17, Algorithm 1) operates in an *optimize and expand* mode, seeking for a plan that maximizes the number of goals that are achieved within a fixed planning horizon. If this number is lower than a user supplied value, PSP extends the planning horizon. The procedure iterates until the number of goals that are achieved reaches the desired value. In order to mitigate the computational burden associated with this optimization task, PseudoSATPLAN adds *intermediate goals* that are elements of the goal set G of the original planning problem, and need to be established at various time points in the plan that will be generated. These goals are introduced as unit clauses in the underlying propositional model of the problem, and through constraint (unit) propagation, yield simpler theories.

Moreover, the optimization part of PseudoSATPLAN is divided into *three phases*, called *initial*, *intermediate* and *final optimization phase* respectively. These phases differ mainly in the way the intermediate goals are generated and used. The basic idea here is that the intermediate goals become increasingly more useful as the plan length increases and, consequently, the corresponding optimization problems that need to be solved become more difficult.

Algorithm 1 PSP-Plan

```
Require: $P =< I, G, A >$: planning problem
    $ip, np, fp$: fraction of goals for each phase
    $iptime, nptime, fptime$: optimization timeout
    $pu$: back stepping parameter
    $strat$: sliding goals parameter
    $impr$: plan improvement flag
    $imprtime$: plan improvement timeout
Return: $plan$
1: init-process($ip, iptime, solution$);
2: $achieved :=$ goals achieved by $solution$;
3: if $|achieved| < \lfloor |G| * np \rfloor$ then
4:    repeat
5:       extend planning graph and build PB problem model;
6:       update-goal-constraints($achieved, strat$);
7:       solvePBO($\lfloor |G| * np \rfloor, nptime, solution, achieved$);
8:    until $|achieved| \geq \lfloor |G| * np \rfloor$
9: end if
10: if $|achieved| < \lfloor |G| * fp \rfloor$ then
11:    repeat
12:       extend planning graph and build PB problem model;
13:       update-goal-constraints($achieved, strat$);
14:       solvePBO($|achieved| + 1, fptime,$
15:             $solution, achieved$);
16:    until $|achieved| \geq \lfloor |G| * fp \rfloor$
17: end if
18: extract $plan$ from $solution$;
19: $plan'$:= initial $\lfloor pu * len(plan) \rfloor$ steps of $plan$;
20: $I'$:= state obtained after executing $plan'$ on $I$;
21: call $SMP(I', G, A, plan'')$;
22: $plan = conc(plan', plan'')$;
23: if $impr$ then
24:    return improve-plan($plan, imprtime$);
25: else
26:    return $plan$;
27: end if
```

Algorithm 2 init-process

```
Require: $ip, iptime$;
Return: $achieved$;
1: run $Graphplan$ until all goals are reachable and non-mutex;
2: repeat
3:     build PB problem model;
4:     solvePBO($\lfloor |G| * ip \rfloor, iptime, solution, achieved$);
5:     if $solution = \emptyset$ then
6:         extend planning graph;
7:     end if
8: until $solution \neq \emptyset$
9: return $achieved$;
```

The *initial optimization phase* (line 1, Algorithm 1) computes a plan that achieves a subset of the problem goals whose size is determined by the value of parameter ip, a real number between 0 and 1 (or percentage). No intermediate goals are used during this phase. This phase is realized by procedure `init-process` which is described in Algorithm 2. It first runs Graphplan until all goals are reachable and non mutually exclusive (line 1, Algorithm 2). It then builds the problem model (line 3), and invokes the main optimization procedure `solvePBO` (line 4) which will be explained later in this section. This procedure is asked, through its first parameter, to find a plan that attains a subset of goals of size at least $\lfloor |G| * ip \rfloor$. If no solution that attains a goal set of the required cardinality exists, the planning graph is extended (line 6) and a new PBO problem is solved. The procedure iterates until a solution that accomplishes the specified number of goals is found.

`PseudoSATPLAN` then enters the *intermediate optimization phase* (lines 2-9, Algorithm 1). First, the goals that are contained in the solution returned by the initial optimization phase of the algorithm, initialize the set of $achieved$ goals (line 2, Algorithm 1). This is a set of temporally annotated facts $g(T)$, where T refers to the time that g must be assigned true in the solution. After the end of the first phase, this time point T is initialized to the current planning horizon. The aim of this optimization phase is to extend the set of goals that are attained (the elements of $achieved$)

Algorithm 3 update-goal-constraints

```
Require: $achieved, strat$;
Return: $achieved$;
 1: if $strat = 0$ then
 2:     for $g(T) \in achieved$ do
 3:         add unit clause $g(T)$ to problem model;
 4:     end for
 5: else
 6:     $achieved' := \emptyset$;
 7:     for $g(T) \in achieved$ do
 8:         if $T_{last}\ mod\ strat = 0$ and $T \neq T_{last}$ then
 9:             $achieved' := achieved' \cup \{g(T+1)\}$;
10:         else
11:             $achieved' := achieved' \cup \{g(T)\}$;
12:         end if
13:     end for
14:     $achieved := achieved'$;
15:     for $g(T) \in achieved$ do
16:         add unit clause $g(T)$ to problem model;
17:     end for
18: end if
```

with new elements until its cardinality reaches $\lfloor |G| * np \rfloor$, where np is an input parameter, again a real number between 0 and 1. As in the previous phase, each time this task fails, the planning horizon is extended (line 5). The difference here is that the elements of $achieved$ become part of the planning problem as *intermediate goals*. They are introduced in the planning problem by procedure `update-goal-constraints`, presented in Algorithm 3. The time point at which they must be true is specified by the value of the input parameter $strat$, that determines the *goal handling strategy*. The fixed goals strategy applies when $strat = 0$ (lines 1-4, Algorithm 3). Then, any solution must assign true to the element of $achieved$ at the time point associated with them when they first entered set $achieved$. Any value for $strat$ different than 0 yields a *sliding goals strategy* (lines 5-18, where T_{last} in line 8 denotes the current planning horizon). According to this strategy, intermediate goals are shifted periodically from their current to the next time point. The frequency of this shift operation is determined by the value of parameter $strat$. Goals are shifted if T_{last} modulo $strat$ is 0. For example if $start = 5$ and the algorithm first entered in the *intermediate optimization phase* at layer 26, the shift operations of goals would occur for planning horizons $T_{last} = 30, 35, 40, \ldots$ etc. Higher values of the $strat$ parameter (slow/infrequent goal shifting) usually lead to better run times, whereas lower values (fast/frequent goal shifting) yield shorter plans. When a shift occurs, the set $achieved$, as well as the problem model, are modified accordingly (lines 14-17, Algorithm 3).

The set of goals that are attained during the optimization part is further expanded in the next, *final optimization phase* (lines 10-17, Algorithm 1). The lower bound on the cardinality of this goal set (the elements of which are in direct correspondence with those of the set $achieved$) for this phase is $\lfloor |G| * fp \rfloor$. The input parameter fp is a real number between 0 and 1, as ip and np parameters. PseudoSATPLAN extends $achieved$ incrementally by requiring that at each time point its size is increased by one (line 14, Algorithm 1). The new goals that are collected along the way are introduced by update-goal-constraints (line 13) to the problem model as dictated by the goal handling strategy. The final optimization phase concludes the optimization part of PSP. The optimization part of PseudoSATPLAN relies to a great extent on the effectiveness of the underlying optimization algorithm that is employed. Therefore, it deserves a closer look.

Procedure solvePBO, outlined in Algorithm 4, optimizes $f^G(\cdot)$ for a given plan length, by repeated calls to precosat solver. The input lower bound on the value of the objective function is expressed via a constraint (line 1, Algorithm 4), and the whole problem translates into a CNF theory by minisat+, that is given as input to precosat (lines 2-3). When a solution that assigns true to m (with $m \geq$ *bound*) of the goals is found, solvePBO attempts to improve it (line 4-11). To this end a new PB problem is solved, using the same technique as before. This time the problem contains the constraint $f^G(\cdot) \geq m + 1$ instead of $f^G(\cdot) \geq$ *bound*. Since at a previous iteration $f^G(\cdot) \geq m$ was solved with a solution $f^G(\cdot) = m$ adding the $f^G(\cdot) \geq m + 1$

Algorithm 4 solvePBO

```
Require: $bound, timeout, achieved$;
Return: $solution, achieved$;
 1: add constraint $f^G(\cdot) \geq bound$ to problem;
 2: run minisat+ to convert PB problem into a SAT theory $T$;
 3: find $solution$ for $T$ by calling precosat;
 4: $solution' = \emptyset$ ;
 5: while $solution \neq \emptyset$ and not $timeout$ do
 6:     update $achieved$;
 7:     add constraint $f^G(\cdot) \geq |achieved| + 1$ to problem;
 8:     run minisat+ to convert PB problem into a SAT theory $T$;
 9:     $solution' = solution$;
10:     find $solution$ for $T$ by calling precosat;
11: end while
12: if $solution \neq \emptyset$ then
13:     return $solution$ and $achieved$;
14: else
15:     return $solution'$ and $achieved$;
16: end if
```

constraint aims at solving a more difficult problem that is finding a better solution in respect to the objective function $f^G(\cdot)$. The procedure iterates until the solution cannot improve further. The input parameter $timeout$ imposes a time limit on the optimization algorithm. No such limit applies to the search for the first solution (line 3). The optimization procedure `solvePBO` is invoked in all the three phases of the optimization part (initial part line 5 Algorithm 2, intermediate and final parts line 7 and 14 respectively, Algorithm 1) of `PseudoSATPLAN`.

For each of the three phases *initial optimization phase*, *intermediate optimization phase*, and *final optimization phase* of the optimization part, the user can provide a time limit (one for each part), *postponeip*, *postponenp* and *postponefp* respectively. (We do not present these in the pseudo code of the algorithm in order to keep the pseudo code as readable as possible). For any of the phases that its termination condition is satisfied, that is the number of goals to achieve according to parameters, if the total time the algorithm spend on that phase is less than the time limit parameter then the algorithm will insist on that phase. For example if *postponenp* is set to 120 seconds by the user and $\lfloor |G| * np \rfloor + 1$ goals are found true in 40 seconds of execution time of *intermediate optimization phase*, the PSP algorithm will remain in the *intermediate phase* although $\lfloor |G| * np \rfloor + 1 > \lfloor |G| * np \rfloor$ for another 80 seconds, trying to satisfy even more goals. The rationale is that if the algorithm spends little execution time in any of the three optimization phases, it might be worthwhile to insist in this phase since is possible to satisfy even more goals before proceeding to the next phase. However we did not use this feature (as are the default parameters in the PSP algorithm) in our experiments, since we did not find any significant advantage in general.

During the implementation of the PSP we experimented with an alternative version of the algorithm. In this version the order of the two later phases of the optimization phase *intermediate optimization phase* and *final optimization phase* are executed in reverse order. The order in this version of the phases of the optimization part are: *initial optimization phase*, *final optimization phase* and *intermediate phase* as these phases where explained before in this section. In fact the PSP system implements this order if the user sets the parameter $optpriority$ to 2. The default value is 1 and implements the optimization part in the 'normal' order *initial optimization phase*, *intermediate phase* and *final optimization phase* as it is presented in algorithm 1. We experimentally found that the default version of the algorithm PSP performs better in all large problems from a variety of domains.

5.3.2 Satisfaction part

The outcome of the optimization part is a solution that achieves at least $\lfloor |G| * fp \rfloor$ of the problem goals. This is the input to the *satisfaction part* (lines 18-22, Algorithm 1), that first extracts an initial $plan$ from the solution (line 18, Algorithm 1). Then, depending on the value of parameter pu, which is called the *back stepping* parameter, computes a new plan that contains the first $\lfloor pu * len(plan) \rfloor$ steps of the original plan (line 19), where len is a function that returns the length of the $plan$. The purpose of this back stepping operation (if considered in conjunction with the next steps of the algorithm in lines 21 and 22) is to slightly mitigate the greedy behavior of the planning procedure. Indeed, PSP tends to generate plans that include actions which lead to the achievement of a subset of the goals, and ignores other actions that could possibly contribute to the achievement of other goals which are not included in this subset. The experimental evaluation has shown that this back stepping technique can improve the quality of the generated plans.

At this point PSP-plan formulates a new planning problem whose initial state is the one that is obtained after the execution of the actions that belong to the part of the plan which has been selected in the previous step of the algorithm (line 20). The final state and the actions remain the same as in the original problem. The new planning problem is solved by the SMP planner (line 21). The idea here is that, for sufficiently high values for the fp parameter, finding a plan for the new problem (using SMP or any other system) is expected to be a task that is substantially easier than that of finding a plan for the original problem. This is because the initial state of the new problem is obtained after the execution of actions which belong to a plan that achieves a subset of the goals. This intuition has been verified experimentally.

The result $plan''$ that is returned by SMP is concatenated with the previous $plan'$, computed at the end of the optimization part of PSP-plan. Recall that $plan''$ is a set of actions that transform state I' to the final state, whereas $plan'$ transform the original initial state into I'. Therefore their concatenation is a first valid plan for the original problem (line 23).

Any of the parts or phases of the planning process can be omitted by selecting suitable values for the input parameters of the PSP-plan algorithm. For instance, by setting $np = fp$ the final optimization phase is omitted.

5.3.3 Improving the solution

If the user so wishes, `PSP` can attempt to improve this first solution $plan$. This task is carried out by procedure `improve-plan` (line 24), which when successful, returns a new improved (i.e. shorter) plan, otherwise it returns the $plan$. Procedure `improve-plan` is implemented by repeated calls to `SMP`, that first searches for a plan of length $len(plan) - 1$. If successful, `improve-plan` then searches for a plan with length $len(plan) - 2$, etc. The process terminates when either the user supplied timeout value $imprtime$ is reached or an optimal plan is found. A plan of length $len(plan) = l$ is proved to be optimal in the obvious way: `SMP` proves that the theory for the planning problem with horizon $l - 1$ is unsatisfiable.

5.4 Experimental evaluation

This section presents the results of a preliminary experimental evaluation of `PseudoSATPLAN`, in domains from various planning competitions. All experiments were run on a server with 24 X5690 cores at 3.47GHz running under CentOS. A CPU timeout of 3600 seconds was used in all experiments, and the values for the optimization timeout parameters $iptime$, $npitme$ and $fptime$ were set to 40, 80 and 80 seconds respectively. In the following, the string X-Y-Z-W denotes a set of `PSP` runs with parameter values $ip = X$, $np = Y$, $fp = Z$, $strat = W$.

Table 5 provides some initial results for `PseudoSATPLAN`, and a comparison with `SMP`. The entries under `SMP` are the number of problems solved in each domain by `SMP` within the timeout limit, whereas the entries under 25-50-50-0 and 25-50-50-1 are the number of problems solved by `PSP` with parameter values $ip = 0.25$, $np = fp = 0.5$ (i.e. the final optimization phase is omitted) and $strat = 0$ and $strat = 1$ respectively. It is interesting that although not at its full strength,

Domain	Problems	SMP	25-50-50-0	25-50-50-1
Depots	22	20	22	22
DriverLog	20	17	18	18
Zenotravel	20	16	19	19
Freecell	20	6	9	9
Satellite	36	18	20	20
Pathways	30	17	27	26
Storage	30	17	20	21
Elevators	30	14	20	20
ScanAnalys	30	20	20	20
Transport	30	13	19	20
Visitall	20	12	16	14
Barman	20	8	15	15
Total	308	178	225	224

Table 5: Number of problems solved by SMP and PSP in different domains. PSP was run with $ip = 0.25$, $np = fp = 0.5$ and $strat$ 0 and 1 (two last columns respectively).

already PSP achieves a remarkable increase of 47 (or about 25%) in the number of problems that are solved.

Table 6 provides a more detailed comparison of the effects of the values of different parameters on the performance of PseudoSATPLAN. Firstly note that increasing the value of the fp parameter up to 70% (i.e. asking that the outcome of the optimization part is a plan that achieves 70% of the goals), with a relative increase of the np value, improves the effectiveness of PSP. Higher values for fp do not seem to result in any gain. Moreover, the optimal value for the search strategy parameter $strat$ depends on the values of the other parameters. In any case, the best values for $strat$ are in the range between 2 and 5. Finally, the best value combination for the parameters of PseudoSATPLAN is 30-50-70-5, for which it achieves an impressive increase of 66 (37%) in the number of solved problems compared to SMP.

Domain	25-50-0-3	30-45-60-0	30-45-60-2	30-45-60-4	30-45-60-5	30-50-70-0	30-50-70-1	30-50-70-2	30-50-70-3	30-50-70-4	30-50-70-5
Depots	22	22	22	22	22	22	22	22	22	22	22
DriverLog	19	18	20	20	19	20	19	19	19	19	20
Zenotravel	19	18	19	19	19	18	19	18	18	18	18
Freecell	9	9	9	9	9	9	9	9	9	9	9
Satellite	20	20	20	20	20	20	20	20	20	20	20
Pathways	26	25	26	25	24	27	25	27	28	26	26
Storage	23	23	24	24	23	26	23	25	27	26	27
Elevators	21	22	23	24	22	23	21	22	23	23	24
ScanAnalys	20	20	20	20	20	20	21	22	22	22	22
Transport	22	18	20	20	19	18	19	19	20	20	20
Visitall	16	16	16	16	15	16	16	16	16	16	16
Barman	19	15	16	19	18	15	15	16	16	20	20
Total	236	226	235	238	230	234	229	235	240	241	244

Table 6: Number of problems solved in 12 domains by PSP with different combinations of parameter values. X-Y-Z-W in the first line denotes a PSP run with parameter values $ip = X$, $np = Y$, $fp = Z$, $strat = W$.

The remaining of this section discusses the issue of the quality of the plans generated by PseudoSATPLAN. The comparison is performed on the parameter value combinations 30-45-60-4, 30-50-70-3, 30-50-70-4, and 30-50-70-5 that yield the highest numbers of solved problems. While these runs were performed with the back stepping parameter set to 1, the result of the runs with $pu = 0.85$ (i.e. 85% of the plan that is returned by the optimization phase is used) for the combination 30-50-70-5 (denoted as 30-50-70-5-PU85) are also reported.

An interesting first result concerns the number of problems for which an optimal solution has been found by PSP. For all the combination of values described in the preceding paragraph, this number is as follows: Barman 8, Depots 19, DriverLog 17, Zenotravel 16, Freecell 6, Storage 17, Elevators 14, ScanAnalys 20, Transport 13, Visitall 12. A comparison of these numbers to the entries of the SMP column of Table 5, leads to the (not very surprising) conclusion that in almost all these 10 domains PSP finds an optimal plan if SMP does.

Domain	30-45-60-4		30-50-70-3		30-50-70-4		30-50-70-5		30-50-70-5-PU85	
Depots	455	308	458	302	450	303	469	305	398	302
DriverLog	300	215	282	216	300	215	307	215	252	215
Zenotravel	153	108	155	108	153	108	153	108	140	108
Freecell	159	137	159	131	159	131	161	130	147	130
Satellite	221	180	233	180	242	180	235	180	211	174
Pathways	596	450	571	444	560	446	580	445	521	444
Storage	300	270	304	271	307	273	325	272	294	267
Elevators	582	532	594	536	594	542	609	556	575	527
ScanAnalys	137	127	166	127	170	127	166	127	166	127
Transport	391	318	388	323	394	327	396	325	384	313
Visitall	525	487	535	487	533	487	538	486	528	488
Barman	781	758	773	748	781	733	790	777	778	696
Total	4600	3890	4618	3873	4643	3872	4729	3926	4394	3791

Table 7: Sums of the lengths of plans generated by PSP with different parameter values. For each run the left column contains the sum of the lengths of the *first* plans that are computed, and the right column the sum of the lengths of the *best* plans.

The only exception is Depots where PSP misses an optimal solution, and this happens irrespectively of its parameter values.

In Satellite, PSP finds the same number of optimal solutions as SMP in most runs, and misses only one optimal solution in the others. The situation is similar in Pathways with the difference that run 30-45-60-4 finds the optimal solution for 13 problems, whereas SMP returns optimal plans for 17.

The effect of the values of the parameters of PSP on the quality of the plan has been also investigated, and the results are depicted in Table 7. The entries of this table refer to the sum of the length of the plans returned for each domain by five different runs of PSP with different parameter values. The first four runs are with $pu = 1$ (the back stepping parameter), whereas for the last run $pu = 0.85$. To be meaningful, the comparison has been performed on problems that are solved in all runs. For each PSP run, Table 7 contains two columns. The left sums the lengths of the *first plans* generated by each PSP run for each problem and domain considered. The right

Domain	30-50 70-5-100	30-50-70-5-85	30-50-70-5-70
Depots	22	21	22
DriverLog	20	19	18
Zenotravel	18	18	19
Freecell	9	9	9
Satellite	20	19	20
Pathways	26	29	28
Storage	27	25	22
Elevators	24	23	20
ScanAnalys	22	22	21
Transport	20	18	19
Visitall	16	16	16
Barman	20	20	17
Total	244	239	231

Table 8: Number of problems solved by PSP with different values for the back stepping parameter. The values for the last three columns, taken from left to right, are $pu = 1$, $pu = 0.85$ and $pu = 0.70$.

column is the sum of the lengths of the *best plans* generated by each PSP run.

A first conclusion is that the plan improvement phase results in a shortening of the plans of about 15%. Moreover, the length of the first generated plan increases with the value of parameter $strat$. Finally, the back stepping technique fulfils its purpose as the plans that it generates are shorter.

As Table 8 reveals, the increased plan quality that comes with lower values for the back stepping parameter, has a negative impact on the number of solved problems. Indeed, for the combination of values 30-50-70-5 the total number of solved problems goes from 244 for $pu = 1$ to 239 for $pu = 0.85$, and further down to 231 for $pu = 0.7$.

Finally, Table 9 compares the plan length and computation time of SMP with various PseudoSATPLAN runs on the hardest problems solved by SMP. The time that PSP needs to generate the first solution is always lower than SMP's cpu time. This difference can be several orders of magnitude (see for instance the problems from the Visitall and Storage domains). The quality of the first solution varies across the domains, but it seems that its length is always less than twice the length

Domain	SMP		30-45-60-4				30-50-70-5				30-50-70-5-PU85			
Depots-22	12	2773	22	1859	21	3600	19	895	17	3600	16	857	13	3600
DriverLog-16	18	477	29	352	18	1490	24	153	18	765	22	205	18	841
Zenotravel-16	7	358	11	140	7	737	11	189	7	857	10	181	7	717
Freecell-5	16	213	20	121	16	326	20	122	16	329	19	124	16	320
Satellite-19	12	605	15	75	12	772	17	107	12	811	15	108	12	738
Pathways-21	24	3041	36	519	24	3600	31	538	24	3545	26	536	24	3450
Storage-17	12	2080	13	6	12	1866	13	11	12	1687	13	15	12	1765
Elevators-21	21	1205	27	151	21	1856	26	164	21	1813	24	178	21	1357
ScanAnalys-9	15	3267	16	118	15	2433	17	29	15	2249	16	25	15	2269
Transport-5	19	802	25	192	19	2885	26	152	19	2984	24	194	19	2669
Visit-6-full	35	2767	38	11	35	2148	39	19	35	2142	37	30	35	2216
Barman-6	41	552	43	20	41	416	43	30	41	351	42	22	41	327

Table 9: Comparison of plan length and cpu time of SMP and PSP on the hardest problems solved by SMP. The columns for SMP contain the plan length (left column) and cpu time. For each PSP run the four columns are as follows: the first two are the length and cpu time for the first plan, and the other two the length and cpu time for the best plan computed by PSP. Value 3600 means that the cpu time limit was reached, and PSP execution was aborted.

of the optimal plan. Moreover, the plan improvement step almost always (with the exception of the problem from the Depots domain) returns the optimal solution within the cpu timeout. In terms of cpu time, there are cases where SMP finds the shortest solution faster, and problems where PseudoSATPLAN outperforms SMP. Finally, as noted before, the back stepping technique can improve the quality of the generated plans.

Therefore, PseudoSATPLAN represents an important advancement over existing implementations of the planning as satisfiability idea. Indeed, PSP has been proven a powerful planning system that is capable of solving hard planning problems by generating parallel plans of high quality.

5.5 Implementation issues of PseudoSATPLAN System

The PSP system is a *proof-of-concept* implementation. There are two major drawbacks of the current implementation of the system.

The first drawback is that the module that builds the planning graph. PSP is build over the SATPLAN framework. More precisely, it is build as an extension of the SMP system [111], which in turn builds on the SATPLAN [76] and BLACKBOX [75] systems. The module responsible for building the graph is the one used by BLACKBOX [75], a system implemented fifteen years ago. Some of the benchmarks in the IPC competitions available today are just too demanding for the BLACKBOX system. For large problems of some domains the CPU time of PSP system is dominated by the construction of the planning graphs for successively larger planning horizons than by the actual search (solving PB constraints translated to CNF clauses) done by the *precosat* solver. There are some ways to solve this problem. The first way is to rewrite the module for building the planning graph from scratch, as in other SAT-based planners, for example Madagascar [101, 102]. Another way is to build the planning graph at a preprocessing step for a large planning horizon x and store it in a file. Then at each iteration for a planning horizon y of the PSP system (where $y \leq x$) this file can be used (constraints corresponding to the time steps equal or less than y with the addition of objective function) to build the PB theory. Such a large planning horizon can be found by running a fast suboptimal planner such as LAMA planner [96, 94], or by simply building a planning graph for a horizon considerably larger than the first layer that the planning graph levels-off. Such ideas are implemented in Maxplan planner [31] described in chapter 3.

The second drawback is the optimization procedure solvePBO described in a previous subsection. Procedure solvePBO optimizes $f^G(\cdot)$ for a given plan length, by repeated calls of the precosat solver. At each iteration the SAT solver does not use anything from the solution found in previous iterations: it tries to find a solution from scratch. Moreover minisat+ and precosat solver use temporary files to communicate for the input problem, objective functions, output solutions and mapping files for the variables. The time spent for manipulating these files for large problems is an addition to the total plan time. To resolve these issues, in a future version of the system the optimization procedure must be rebuild. As was described before, minisat+ has as an underlying solver the outdated minisat solver. Since both minisat+ and precosat solvers are open source systems, it is possible for the minisat+ system to be altered to use precosat solver (or any other open source code state of the art SAT solver at time) as the underlying solver. In this way the use of temporary files is not needed and, more importantly, all the learnt clauses found at a previous iteration by the SAT solver can be used at a next iteration. We believe that this could speed up the optimization procedure.

Another idea we want to investigate in a future implementation, is the possibility to use learnt clauses found while solving for a planning horizon, in the successive encodings for larger planning horizons. Similar methods are implemented in the MAXPLAN planning system [30] that was briefly presented in chapter 3 section 3.3.3. In this way the solver can use a learnt clause for constraint propagation without having to re-discover it, and this may speed up the search. Obviously this must be done without affecting the soundness of the algorithm. For example consider two SAT theories Θ_1 and Θ_2. Assume that a conflict driven clause learning (CDCL) SAT solver, for example precosat, searches for a solution for Θ_1 and learns a clause l. There is a unique traceable set of clauses C such that $C \subseteq \Theta_1$ and C being the reason for the learnt clause l. Obviously if it holds that $C \subseteq \Theta_2$ then l must be a learnt clause (a no-good) in Θ_2 as well, and we can soundly solve $\Theta_2 \cup \{l\}$ instead of Θ_2, since both the theories are logically equivalent. For the SMP encodings for a planning problem for two different planning horizons the clauses are the same up to the layer of goals of the smaller planning horizon minus one. There are two key issues however that must be considered: The translation of the planning graph is done to PB inequalities and then to SAT. Therefore we must ensure that minisat+ translates a PB variable for the two different planning horizons to the same SAT variable, otherwise we will need to map the SAT variables for the two planning horizons. The other issue is the sliding goals strategy. Assume that for a planning problem PSP system builds two SAT theories Θ_1 and Θ_2 for planning horizons x and y ($y > x$) respectively. Also assume that there is a set C of unary clauses of intermediate goals at a layer k such that $C \subset \Theta_1$. Since the goals 'slide' it may hold that $C \not\subset \Theta_2$ (because PSP slided C to a layer t, $k < t < y$). Hence it is unsound to put a learnt clause in Θ_2 if a clause in C was used by the SAT solver when using Θ_1 for its computation.

5.6 Conclusions

This chapter presented planning as (pseudo-boolean) optimization, a new method for propositional STRIPS planning, that has been implemented in a system called PSP. PSP is capable of solving planning problems that are beyond the reach of current implementations of the SATPLAN approach (for optimal planning). Although it cannot guarantee the optimality of the solutions it returns, PSP is not far behind those systems in terms of the number of problems that it can solve optimally. Possible future developments in SAT solving or the modeling of planning as a SAT problem can be directly imported into PSP and enhance its performance.

Chapter 6

Heuristic Guided Optimization for Propositional Planning

The PSP-H planning system

Planning as Satisfiability is an important approach to Propositional Planning. A serious drawback of the method is its limited scalability, as the instances that arise from large planning problems are often too hard for modern SAT solvers.

This work tackles this problem by combining two powerful techniques that aim at decomposing a planning problem into smaller subproblems, so that the satisfiability instances that need to be solved do not grow prohibitively large. The first technique, *incremental goal achievement*, turns planning into a series of boolean optimization problems, each seeking to maximize the number of goals that are achieved within a limited planning horizon. This is coupled with a second technique, called *heuristic guidance*, that directs search towards a state which satisfies all goals. Heuristic guidance is based on the combination of a number of constraint relaxation techniques of varying strength.

Initial experiments with a system which implements these ideas demonstrate that enriching propositional satisfiability based planning with these methods delivers a competitive planning algorithm that is capable of generating plans of good quality for challenging problems in different domains.

6.1 Introduction

Planning as Satisfiability [74, 76], that was presented in detail in previous chapters (chapter 3 and 4), is an important method for optimal propositional planning. The main idea is to translate a planning problem into a series of propositional satisfiability instances, which are then solved by a SAT solver. One of the advantages of the method is that it finds plans of optimal makespan. Despite the recent progress in SAT solving, and the development of strong propositional satisfiability models for planning [29, 106, 61] presented in chapter 4, it often happens that the instances generated from planning problems, grow so large that they are unsolvable even by the most advanced SAT solvers.

This motivated a number of recent attempts to improve scalability at the cost of sacrificing the optimality of the generated solutions. One such example is `Madagascar` [102], that enhanced a SAT solver with a planning specific variable selection heuristic, to generate suboptimal plans quickly. `Madagascar` , that was briefly presented in section 3.3.4, extended the reach of the planning as satisfiability framework by a large margin.
In a different spirit, the `PSP` system presented in chapter 5, introduced the idea of *planning as optimization*. `PSP` employs the increasing planning horizon technique, but unlike classic `SATPLAN`, it maximizes the goals that are achieved within each horizon. In order to restrict the search space, and therefore mitigate the computational burden, `PSP` adds goals that are achieved along the way as intermediate goals in the problem model.

However, as the plan length can grow arbitrarily long, at some point it is expected that the approaches of `Madagascar` and `PSP` will be confronted with performance difficulties. In the work presented in this chapter we take a different path. The basic idea is to split a planning problem into smaller subproblems, of a size that is not prohibitive for the underlying solver. It turns out that the new method is capable of synthesizing plans that are more than two hundred steps long.

`PSP-H`, the new planning system that is described in this chapter, shares with `PSP` the planning as optimization perspective, but it differs in a number of important ways. The first is the *incremental goal achievement* which, at a high level, works as follows. `PSP-H` first generates a sub-plan that, starting from the initial state, achieves a subset, of predefined size, of the problem goals. The state that results after the execution of the actions of this sub-plan, called an *intermediate state*, becomes the new initial state, and a new sub-plan that satisfies a larger subset of goals is computed. The procedure iterates until all goals are satisfied. It then links together the generated sub-plans to produce a solution to the original problem. Therefore, instead of solving the original planning problem, `PSP-H` solves a series of smaller subproblems.

The problem with this greedy approach is that it focuses on maximizing the number of achieved goals in a restricted planning horizon, and it ignores completely goals that cannot be achieved within this horizon. In order to overcome the limitations that this would place on the effectiveness of the system, `PSP-H` is enhanced by a second technique called *heuristic guidance*, which imposes an additional requirement on the intermediate states that are computed by the `PSP-H` algorithm. The property that these states need to satisfy is that there must be a *relaxed* plan from each such state to the final state. `PSP-H` employs three different relaxation methods that are all based on ignoring some of the problem constraints, but they differ in their strength.

The first relaxation method is the well-known delete lists relaxation, whereas the second method ignores all action mutexes. The last relaxation, which is stronger than the first but weaker than the second, ignores action mutexes and uses a subset of the fact mutexes that are heuristically selected. Moreover, for certain classes of fact mutexes $x_1, \ldots, x_n$, instead of the constraint $\sum_{i=1}^{n} x_i <= 1$, PSP-H adds the weaker inequality $\sum_{i=1}^{n} x_i <= k$, for values of k that are explained later.

PSP-H is implemented on top of the PSP system. Therefore, it combines the new ideas introduced in this work with those of chapter 5, to deliver a competitive planning system. Indeed, our experimental evaluation on a number of domains taken from planning competitions, demonstrates that PSP-H can solve challenging problems. Moreover, a preliminary comparison with Madagascar and LAMA, a top performer in the last planning competition, shows that PSP-H contributes viable ideas in the direction of bridging the performance gap between satisfiability based and heuristic planners. We stretch that unlike LAMA, but similarly to Madagascar, PSP-H can generate plans with parallel actions, a desirable feature in some application domains.

The experimental comparison of PSP-H with Madagascar shows that the new system can solve more problems in some domains, and generate better quality solutions both in term of plan length and number of actions. In fact, an important feature of PSP-H is that it can be directed, through suitable parameters, either towards finding solutions fast or in the direction of generating good quality plans.

It is important to note, that the techniques used in PSP-H and Madagascar are not exclusive of each other. For instance, it seems possible to employ the ideas of Madagascar in the optimization step of PSP-H in order to improve its performance in solving the subproblems.

6.2 Planning as satisfiability and Pseudo-boolean optimization

Again our analysis in this work is only for STRIPS planning problems. Recall that a (STRIPS) planning problem P is defined as a triple $P =< I, G, \mathcal{A} >$, where I is the set of facts that hold in the initial state, G is the set of goals, and A is a set of actions. Each action $A \in \mathcal{A}$ has preconditions, add effects and delete effects denoted by $pre(A)$, $add(A)$, and $del(A)$ respectively.

We remind that a *(Linear) Pseudo-boolean constraint* (PB-constraint) is an inequality of the form $\sum a_i x_i \geq b$, where x_i is a boolean variable, and a_i, b integers, as presented in Chapter 2, and a PB-constraint is the generalization of a clause. The propositional clause $x_1 \vee \ldots \vee x_m \vee \neg x_{m+1} \vee \ldots \neg \vee x_n$, where each x_i is an atom, translates into the PB-constraint $x_1 + \ldots + x_m - x_{m+1} - \ldots - x_n \geq m - n + 1$.

The planning as satisfiability framework was presented in detail in previous chapters (chapter 3 and 4) and the pseudo-boolean optimization method of PSP planner was presented in chapter 5. In PSP-H we follow the same translation of the SMP encoding (presented in chapter 4) to PB constraints as in the PSP planning system presented in section 5.2 of chapter 5.

6.3 Heuristic Guidance in Optimization

The incremental goal achievement technique of PSP-H is an extension of the planning as optimization idea of PSP. The main computational task that is carried out by PSP-H is the solution of a PBO problem, where the maximization objective is a function on fact variables associated with the goals of the problem. For simplicity, assume for the moment, that the objective function is the one used in PSP i.e. $g_1(T) + \ldots + g_k(T)$, for $g_1, \ldots, g_k$ goal atoms and T the planning horizon. PSP-H is also supplied with an objective function value threshold $thres$. It starts with an initial planning horizon o that is determined by information that is extracted from the underlying planning graph, as in the classic SATPLAN system.

PSP-H solves the PBO with horizon o and obtains an optimal value f for the objective function. If no solution exists, the planning horizon is set to $o + 1$, and the procedure repeats. If for some horizon o_b a solution is found with a value f_b for the objective function such that $f_b \geq thres$, a $plan$ is extracted from the solution. Unlike PSP, the *intermediate state* I' that results after the execution of the actions in the $plan$ becomes the new initial state, and PSP-H repeats the process with a new planning problem $P' =< I', G, \mathcal{A} >$, where $P =< I, G, \mathcal{A} >$ and I are the original problem and initial state respectively.

Finally, the objective value threshold $thres$ is increased by an amount that is determined by the value of the i_{th} element of parameter $restart\_rate\_array$, where $i - 1$ is the number of sub-problems solved at the time. The parameter $restart\_rate\_array$ is a user supplied array of unique values in the open real interval $(0, 1)$ sorted in ascending order.

As the maximization objective is defined on the fact variables that correspond to goals, the increase in the threshold translates into an increase in the number of goals that are achieved. Hence the term *incremental goal achievement*. The process terminates when the value $thres$ exceeds another user supplied value $finphase$. The output of the above procedure is the concatenation of the partial plans that are generated. Then PSP-H, similarly to PSP, enters the satisfaction part, where the planning problem for the new initial state is solved by SMP. The returned plan, as in PSP, is the concatenation of the plans in the optimization and satisfaction parts.

The periodic replacement of the initial state of the planning problem described above, is called *restart strategy*. With the restart strategy PSP-H attempts to limit the length of the planning horizon, and hence the size of the satisfiability instances that are solved by the system. For instance, $restart\_rate\_array = \langle 0.25, 0.5, 0.75 \rangle$ results in a new restart each time PSP-H finds a state that satisfies 25% more goals than the previous intermediate state, whereas $restart\_rate\_array =$

$\langle 0.2, 0.4, 0.6, 0.8\rangle$ means that the restarts in this case occur as soon as a new state is found, with the number of attained goals 20% higher than that of the previous state. As the length of the plans, and typically the related computational burden, usually increases with the number of goals, relative low values for the differences of each element of the $restart\_rate\_array$ from the next element of the array usually translate into smaller problems and therefore better run times.

As it is often the case with greedy search methods, a difficulty with incremental goal achievement is that it tends to generate locally optimal plans, i.e. plans that maximize the number of attained goals in the limited planning horizon, and ignores goals that are not achievable within that horizon. This bias may lead to poor quality solutions, as well as insolvability when actions interact in complex ways, as in planning with non-renewable resources.

To tackle this problem, PSP-H combines incremental goal achievement with a heuristic method that is intended to account for the effort that is needed to achieve the remaining goals. This is accomplished by a problem model that consists of the usual layers of action and fact propositions that correspond to different times, but it is divided in two parts: the *constrained* and the *relaxed* part.

The constrained part is the standard PB model of the problem, i.e. a translation of the SMP encoding into linear inequalities, as described in section 5.2. The relaxed part is also obtained from the SMP encoding, but some of the constraints of the model are omitted. Moreover, the relaxed part is further divided into three subparts that differ in the mutual exclusion information they contain.

- The *action relaxation* part is the SMP model without action mutex clauses, i.e. without the set of constraints 7 as specified in section 4.2.1. This model is similar to the "$\exists$ encoding" of [103].

- The *intermediate or fact relaxation* is a new type of relaxation introduced in this work. It is less constrained (more relaxed) than the previous part, but more constrained (less relaxed) than the next full relaxation part. As in the action relaxation, no action mutexes are present here. Moreover, for each set of pairwise mutual exclusive facts, either the associated constraint is completely omitted, or a relaxed form of mutual exclusion is added, called *weak exclusion constraint*. Specifically, for a set of pairwise mutex facts $f_1, \ldots, f_k$ and time point T, the corresponding weak exclusion constraint is a cardinality constraint of the form $f_1(T) + \ldots + f_k(T) \leq N$, where $k > N \geq 1$. Note that this is a strict relaxation of the mutual exclusion $f_1(T) + \ldots + f_k(T) \leq 1$ in the case that $N > 1$. More details on the construction of the intermediate relaxation part are provided in the subsection 6.3.1.

- The *full relaxation* part contains neither action nor fact mutexes, i.e. both sets of clauses 7 and 8 are omitted. The clause sets 4 and 6 are also omitted as redundant. This part corresponds to the well-known delete-lists relaxation heuristic that forms the basis of many heuristic search planners.

The encoding is depicted graphically in Figure 4. It starts with the layers of the constrained part, followed by the action relaxation and then the intermediate relaxation part, and ends with the full relaxation layers. Goals are added as unit clauses in a final layer $q + 1$ (layer numbering is taken from figure 4).

The fact variables that are associated with time point k and assigned true in a solution of a problem, define a state s_k. Obviously, s_k can be reached from the initial state by a plan of length k, and satisfies as many goals as possible. On the other hand, the final state is reachable from s_k in $q + 1 - k$ *relaxed* steps. Therefore, while we seek to maximize the number of goals that are satisfied in s_k, at the same time we require that a relaxed plan of length $q + 1 - k$ exists from state s_k to a state that satisfies all goals. Therefore, the heuristic part guides search towards states that attain all goals. Hence the term *heuristic guided optimization*. Moreover, this strategy is reinforced by shortening gradually the length of the relaxed part as we approach closer to a final state. The next section describes how these ideas are implemented in the PSP-H planner.

The relaxed part is further exploited in the objective function of PSP-H which is more elaborate than that of PSP. For a planning problem with the set of goals G, the objective function is

$$max : w_1 * (g_1(k) + \ldots + g_{|G|}(k)) +$$
$$w_2 * (g_1(m) + \ldots + g_{|G|}(m)) + w_3 * (g_1(n) + \ldots + g_{|G|}(n))).$$

Variable $g_i(k)$ is true if goal g_i is true at time k, the last layer of the constrained part. Similarly, variables $g_i(m)$ and $g_i(n)$ refer to the last layer of the action relaxation and intermediate relaxation respectively. The coefficients w_1, w_2, and w_3 are integer values provided by the user, who may wish to assign ”importance” to the goals in different parts of the model. For instance a user may want to implement a search strategy that favours goal achievement in the relaxed part. These three values compose the weight vector in Algorithm 5, where it is denoted by $weight\_vect$. Any of these values can be assigned the value 0.

Finally, it is very important to note that PSP-H overall approach relies on finding intermediate states from which a final state is reachable by a relaxed plan. However, the existence of a relaxed plan does not necessarily imply that there is a sound plan from the intermediate state that achieves the goals of the problem. Indeed, the current version of the system does not employ any method that is able to identify intermediate states that are *deadlocks* or *dead-ends*, i.e. states from which the goals are unreachable. To mediate the problem, PSP-H resorts to outside assistance, namely the Torchlight tool [56]. More specifically, before starting, PSP-H invokes Torchlight

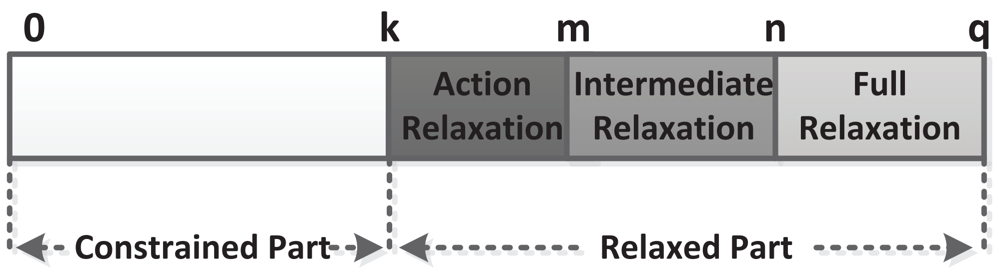

Figure 4: The different part of the SAT encoding in `PSP-H`.

that searches for dead-end states (with parameters `-s 500 -d 20`). If no dead-end state is found (e.g. `Torchlight` returns `Dead-end states:` 0%) the domain is considered *safe*, otherwise *unsafe*. For unsafe domains, `PSP-H` can employ the relaxation part more cautiously, as explained in a following section.

6.3.1 Fact Constraint Relaxation

Fact constraint relaxation is one of the relaxation methods employed by `PSP-H` and implemented in the intermediate part. Its construction is carried out as follows. `PSP-H` first invokes the preprocessor of `LAMA` [96] to generate the multi-valued state representation of the problem. All variables with less that 3 values in their domains are excluded from any further consideration. For the rest of the variables, and from the information provided by `LAMA` preprocessor, `PSP-H` computes for each variable v the shortest path (in terms of actions) between any pair of variable values, and stores this information in the structure $distances(v)$. Hence, $distances(v)[x_i, x_j]$ is the minimum number of actions that are required for the values of v to change from x_i to x_j. If for some variable v there is a pair of values such that there is no path from one to the other, this variable is marked as a *deadlock variable*.

When the user invokes `PSP-H`, she may require, through a suitable parameter value, that problems that are determined unsafe by `Torchlight`, are treated more cautiously, in a sense explained at the end of this section. For safe domains, `PSP-H`, based on the distance information, first identifies the *counter variables*. A variable with n different values is a counter variable if exactly one pair of values have distance $n-1$, two pairs of values have distance $n-2$, etc. For each counter variable v with values $x_1, \ldots x_n$, `PSP-H` adds to the problem model the inequality $\sum_{i=1}^{n} x_i <= 1$. In the current version of `PSP-H`, in the presence of counter variables, no other constraint is added to the intermediate relaxation part.

For safe domains that do not contain counter variables, `PSP-H` computes for each variable v the maximum value of $distances(v)[x_i, x_j]$ over all values x_i, x_j of v. If this value is less than 4, the variable is removed. If no variables remain, `PSP-H` discards the intermediate relaxation part all together, and, in the current implementation, aborts execution if the user enforced the inclusion of that part through parameter values that are explained later.
Otherwise, for each remaining variable v with values $x_1, \ldots, x_n$ `PSP-H` add to the intermediate part the constraint $\sum_{i=1}^{n} x_i \leq k$, where k is 3 if $n <= 100$ and 4 otherwise.

For unsafe domains, the user decides whether they are treated by `PSP-H` in the same way as safe, or differently. In the alternative handling of unsafe domains, the length of the full relaxation part is set by the system to one. Moreover, a mutex inequality of the form $\sum_{i=1}^{n} x_i \leq 1$ is added to the intermediate relaxation part, where the x_i's are the values of a deadlock variable. Finally, a similar constraint is added on the values of each variable that appears in an action that changes the value of the deadlock variable. When in this configuration, we say that `PSP-H` is in the *unsafe mode*.

6.4 The PSP-H algorithm

This section provides some more insight in the internal working of the system. The PSP-H algorithm, presented in Algorithm 5, is the main procedure of the implemented planning system. It takes as input a (STRIPS) planning problem $problem = (I, G, \mathcal{A})$ and a number of parameters, listed in the Require statement of the algorithm, and returns a plan. PSP-H algorithm operates in three parts, the preprocessing, optimization and satisfaction parts.

6.4.1 Preprocessing part

The main task of the preprocessing part (lines 1-3, Algorithm 5) is carried out by procedure GraphPlan-relaxed (line 1). The procedure builds a planning graph for the relaxed version of the input planning problem obtained by removing the delete-lists of all actions of the original problem. The graph is expanded until all goals become reachable. As there are no delete effects, finding a plan for the relaxed problem is an easy task. Based on the length of this plan and the user supplied value of the parameter h_init_perc, PSP-H computes (line 2) the value of variable h_len, which determines the length of the full relaxation part (corresponding to value $q - n$ in figure 4) of the problem encoding.

6.4.2 Optimization part

The optimization part (lines 4-20 Algorithm 5) is the core of the PSP-H Algorithm. In this part the problem is decomposed, using the incremental goal achievement and restart techniques, into a series of subproblems which are successively solved by procedure *optimize* (line 6,

Algorithm 5 PSP-H

```
Require: $problem = (I, G, A)$, $finphase$, $h\_init\_perc$, $h\_red\_rate$, $r\_init\_perc1$,
    $r\_init\_perc2$ , $reduce\_r\_red\_rate$, $weight\_vect = \langle w_1, w_2, w_3 \rangle$, $restart\_rate\_array$;
Return: $solution$;
 1: GraphPlan-relaxed($problem$,$plan$);
 2: $h\_len := \lfloor$length($plan$) $* h\_init\_perc\rfloor$;
 3: $current\_best$:=0;   $problem\_initial = problem$;   $subplan$:=0;   $I'$:=$I$;  $G'$:=$G$;
      $G_{init}$:=$G$;
    $Plan_1$:=<>;   $previous\_f\_achieved\_score$:=0;
    $achieved\_score$:=0;   $r\_red\_rate$:=1;
 4: while $true$ do
 5:   compute_curr_opt ($problem, G_{init}, weight\_vect,$
                 $restart\_rate\_array[subplan + 1], finphase,$
                 $achieved\_score, boundopt, finscore$);
 6:   optimize($problem, boundopt, h\_len, r\_init\_perc1,$
                $r\_init\_perc2, r\_red\_rate, weight\_vect, plan$);
 7:   $achieved\_score$ :=achieved($plan$);    $subplan := subplan + 1$;
 8:   $Plan_{subplan} := plan$;
 9:   $I'$:= state obtained after executing $plan$ on $I'$;
      $G' := G' \setminus \{g | g \in I' \cap G'$  $and$  $\nexists a \in A$  $such\ that$  $g \in del(a)\}$;
      $problem := (I', G', A)$;
10:   if $achieved\_score \geq finscore$ then
11:     exit while
12:   end if
13:   $fraction\_achieved\_score := \frac{(w_1+w_2+w_3)*(|G_{init}|-|G'|)+achieved\_score}{(w_1+w_2+w_3)*|G_{init}|}$
14:   if $h\_len = 1$ and $fraction\_achieved\_score > previous\_f\_achieved\_score + h\_red\_rate$
      then
15:     $r\_red\_rate := r\_red\_rate * reduce\_r\_red\_rate$;
        $previous\_f\_achieved\_score := fraction\_achieved\_score$;
16:   end if
17:   if $h\_len > 1$ and $fraction\_achieved\_score > previous\_f\_achieved\_score + h\_red\_rate$
      then
18:     $h\_len := h\_len - 1$;
        $previous\_f\_achieved\_score := fraction\_achieved\_score$;
19:   end if
20: end while
21: FinalPhase($problem\_initial$,$concat(Plan_1, \ldots, Plan_{subplan})$, $solution$);
```

Algorithm 5), presented in Algorithm 7, and discussed in a following paragraph. The decomposition of the problem is controlled by the user supplied array parameter $restart\_rate\_array$, that has been introduced in the previous section.

The optimization part starts with a call to procedure $compute\_curr\_opt$ (line 5), presented in algorithm 6, that returns two bounds, $boundopt$ and $finscore$. The former, $boundopt$, is used as a lower bound of the value of the objective function that must be returned by the call of the procedure $optimize$ (line 6), and the latter, $finscore$, is used as the threshold for the algorithm to exit the optimization part (lines 10-11) and proceed to the satisfaction part (line 21).

Procedure $compute\_curr\_opt$ takes as inputs values for the parameters $weight\_vect$ =$\langle w_1, w_2, w_3 \rangle$, and $finphase$. The $weight\_vect = \langle w_1, w_2, w_3 \rangle$ corresponds to the weights of the objective function in page 150, and $finphase$ is a number in the open real interval $(0, 1)$ which is used in the computation of $finscore$ (line 7). The other input parameters of $compute\_curr\_opt$ are $problem$, G_{init}, $restart$ and $achieved\_score$. The parameters $problem$ and $achieved\_score$ are the current subproblem to be solved (line 9 of algorithm 5) and the score of the solution of the previous subproblem (line 7 of algorithm 5) respectively. The parameter $restart$ is a number in the open real interval $(0, 1)$, which is used in the computation of the lower bound $boundopt$. In fact, $restart$ is the user supplied ratio score for the next restart (line 5, algorithm 5) of the main algorithm. Parameter G_{init} is the set of goals in the initial problem. Note that $G_{init} \supseteq G$, where G is the set of goals in the $problem$. It may also be the case that $G_{init} \supset G$ because at each restart, any goal that is in the initial state of the new subproblem but not in the delete effects of any action of the problem can be removed from the goal list, as it cannot be 'undone' again (line 9 of algorithm 5). An example is the $visitall$ domain where a robot, that starts in the middle of a square two dimensional grid, must visit at least once all the squares of the grid. This type of goals must not be taken into account in the objective function since they will not be considered again, hence the subtraction of the term $W\_G' - W\_G$ in lines 3,5 and 7 in algorithm 6. If the score found in the previous solution in variable $achieved\_score$ is equal or greater than $W\_G' * restart$, then the next lower bound to restart is updated to $achieved\_score + 1 - (W\_G' - W\_G)$, otherwise it is updated to $W\_G' * restart - (W\_G' - W\_G)$ (lines 2-5 algorithm 6).

Algorithm 6 Procedure `compute_curr_opt`

```
Require: $problem = (I, G, A), G'(\supseteq G), weight\_vect = \langle w_1, w_2, w_3 \rangle,$
         $restart, finphase, achieved\_score$
Return: $boundopt, finscore$;
1: $W\_G := (w_1 + w_2 + w_3) * |G|$;
   $W\_G' := (w_1 + w_2 + w_3) * |G'|$;
2: if $subplan > 1$ and $achieved\_score \geq W\_G' * restart$ then
3:     $boundopt := achieved\_score + 1 - (W\_G' - W\_G)$;
4: else
5:     $boundopt := W\_G' * restart - (W\_G' - W\_G)$;
6: end if
7: $finscore := W\_G' * finphase - (W\_G' - W\_G)$;
8: return $boundopt, finscore$;
```

Procedure $compute\_curr\_opt$ (line 5) described in the previous paragraph calculates the $boundopt$ which is the lower bound passed to procedure $optimize$. Procedure $optimize$ returns a plan (line 6) and the value of $achieved\_score$ is updated with respect to this plan (line 7). The new plan is a sub-plan (line 8) of the final solution that is synthesized by the algorithm in the $FinalPhase$ (line 21). After the (sub)plan is found, `PSP-H` determines the state I' that is obtained by executing the actions of this plan (line 9). Moreover any goals in the goal set that are in I', and there is no action in the problem that deletes them, are removed, as they cannot be falsified (line 9). The new goal set is G'. At this point `PSP-H` applies the restart strategy by replacing the previous initial state with the new state I' and the previous set of goals with G' in the new problem (line 9) that will be solved in the next iteration of the optimization loop.

If the current value of the objective value reaches the value of variable $finscore$, computed by procedure $compute\_curr\_opt$ (line 5), `PSP-H` exits the while loop (lines 10-12) and moves to the `FinalPhase` (line 21) of the *satisfaction part* explained in the following subsection. Otherwise `PSP-H` returns to the top of the optimization loop, procedure $compute\_curr\_opt$ computes a new lower bound that is passed to procedure $optimize$, and the process iterates. Since the bound for a subproblem is strictly greater than the score achieved by the solution of the previous subproblem, successive solutions of subproblems converges towards a solution of the original problem.

As discussed previously, it is sensible that `PSP-H` uses the estimation provided by the heuristic on the relaxed part more cautiously as the number of the remaining unsatisfied goals decreases. `PSP-H` shortens the relaxed part dynamically as the planner approaches a final state. A (user-defined) h_red_rate percentage threshold is used to shorten the relaxed part of the plan. If the length of the full relaxation part (variable h_len) is greater than 1, it leads to a reduction by one step of the length of the `full relaxation part`, whenever the ratio of the objective function of the solution to the global optimal in the current sub-problem (calculated at line 13) is h_red_rate percent greater than the corresponding ratio of the solution found at the previous reduction of the relaxed part (lines 17-19). However, if the condition holds for the score of the objective function and h_len is 1, the full relaxation part cannot be reduced further and the (user-defined) $reduce\_r\_red\_rate$ percentage is used to reduce the number of the layers of `intermediate relaxation part` of relaxed part (lines 14-16).

Each subproblem is an optimization problem that is solved by procedure $optimize$ (line 11, Algorithm 5) presented in Algorithm 7. The procedure maximizes the score of the objective function (or heuristic score) within a user-defined time bound in a minimal number of plan layers for the input subproblem $problem$, subject to the constraint that the score of the solution is equal or greater than its input parameter $bound$. Its parameters are h_len, r_init_perc1, r_init_perc2, r_red_rate and $weight\_vect$.

Algorithm 7 Procedure `optimize`

```
Require: $problem, bound, h\_len,$
    $r\_init\_perc1, r\_init\_perc2, r\_red\_rate, weight\_vect;$
Return: $plan;$
 1: Extend planning graph until all goals are reachable and not mutex;
 2: $pg\_len :=$ number of layers of planning graph;
 3: repeat
 4:   Translate planning graph into PSEUDOBOOLEAN constraints problem $S$;
 5:   Append $\lfloor pg\_len * r\_init\_perc1 \rfloor$ layers of partially relaxed layers with all fact mutexes;
 6:   Append $\lfloor pg\_len * r\_init\_perc2 * r\_red\_rate \rfloor$ layers of partially relaxed layers with weak
      exclusion constraints over facts;
 7:   Append $h\_len$ layers with no fact and action mutexes;
 8:   Add a layer of facts containing goals as unit constraints;
 9:   Convert layers into PSEUDOBOOLEAN constraints and append to $S$;
10:   formulate-obj-fun($weight\_vect, OBFU$);
11:   solve($S, OBFU, bound, solution$);
12:   if $solution = \emptyset$ then
13:     Extend planning graph by one layer;   $pg\_len := pg\_len + 1$;
14:   end if
15: until $solution \neq \emptyset$
16: Extract $plan$ from $solution$
17: Return $plan$;
```

The procedure builds a planning graph until all goals are reachable and there is no pair of goals that is marked as mutex. It then translates the pg_len layers of this graph into a PseudoBoolean constraint problem using the `SMP` encoding (lines 1-4, Algorithm 7). Then the layers of the relaxed part (figure 4) according to the (user defined) parameters h_len, r_init_perc1, r_init_perc2, and r_red_rate are added from layer $pg\_len + 1$. First the $\lfloor pg\_len * r\_init\_perc1 \rfloor$ layers of the *action relaxation* part are added (line 5), then $\lfloor pg\_len * r\_init\_perc2 * r\_red\_rate \rfloor$ layers of *intermediate relaxation* (line 6) and then h_len layers of *full relaxation* (line 7). The parameter r_red_rate is the one that is reduced dynamically in `PSP-H` main algorithm (Algorithm 5 line 15) as explained in a previous paragraph, hence the added layers of *intermediate relaxation* are reduced when r_red_rate is reduced. Finally, another layer is added containing the goals as unit constraints (line 8). The added layers are converted to PseudoBoolean constraints and appended to the initial problem S (line 9). The appropriate objective function (as it is presented in page

151) is formulated using the user defined $weight\_vect$ as was described in a previous section, and the procedure $solve$ is invoked to solve the problem (lines 10-11). The procedure either finds a valid plan for the problem with objective score equal or greater than the bound, or it fails. If procedure $solve$ finds a first solution it iteratively tries to find a solution with a better score until an optimal solution is found or a (user defined) time limit is reached. If a solution is found, a plan is extracted and returned (line 16-17), else the planning graph is extended by one layer and the whole process repeats. As in the `PSP` planning system presented in chapter 5, the loop for optimizing the objective function of procedure $solve$ is done by successive calls of the `minisat+`[41] to convert the PseudoBoolean constraints to CNF and then solving the problem by invoking the SAT solver `precosat`[16]. The procedure is the same as the $solvePBO$ of `PSP` system presented in chapter 5, section 5.3.1 algorithm 4, with the only difference being the formulation of the objective function.

6.4.3 Satisfaction part

The `optimization` part (line 4-20 Algorithm 5) halts when a solution is found with the score of the objective function being at least $finscore$, where $finscore$ is computed by function $compute\_curr\_opt$ (algorithm 6) as explained in the previous section. The $plan$ found by the *optimization part* is the concatenation of all the sub plans, found i.e.
$plan = concatenate(Plan_1, \ldots, Plan_{subplan})$.

Then $plan$ is passed as a parameter to procedure $FinalPhase$ (as in `PSP` planner) of the `satisfaction part` (line 21 Algorithm 5) which is presented in Algorithm 8. The $plan$ is executed on the initial state I of the original problem to obtain a new initial state I' (line 2 Algorithm 8). If not all goals are achieved in I' the SMP planner is invoked to find a plan $plan'$ for the planning problem from the new initial state I' that satisfies all goals of the original planning

Algorithm 8 Procedure `FinalPhase`

```
Require: $problem = (I, G, A), plan$
Return: $solution$;
1: $plan' := <>$;
2: $I'$:= state obtained after executing $plan$ on $I$;
3: if $achieved < |G|$ then
4:     SMP($(I', G, A), plan'$);
5: end if
6: $solution := concat(plan, plan')$
7: return $solution$;
```

problem (lines 3-5). Finally the concatenation of $plan$ and $plan'$ is the returned solution (lines 6-7) of the input planning problem that is the output of the `PSP-H` planner.

6.5 Experimental evaluation

`PSP-H` is a prototype system that has been implemented to test the techniques that are presented in this work. Similarly to `PSP`, it has been built on top of `SATPLAN`, from which it inherits the very slow and memory demanding planning graph construction phase. This imposes serious limitations on the size of the problem that can be solved by `PSP-H`. In fact, as we note later, in many domains the current version of `PSP-H` spends more time in manipulating data structures than in SAT solving.

`PSP-H` has been evaluated on a number of domains from various planning competitions and compared with `LAMA` and `Madagascar`. We stretch here that `PSP-H` has been given an advantage over the other systems (especially `LAMA`) in this comparison. Indeed, in some domains, problems which `PSP-H` cannot solve because of high memory demand, are excluded from the comparison. While it seems that the other system can solve some of the the excluded problems, it is uncertain that `PSP-H` would be able to solve them, were not memory consumption an obstacle. Moreover, these experiments are not suitable for comparing the performance of the other two planners. Nevertheless, our results clearly indicate the advantages and the viability of the PSP-H approach in certain domains.

All experiments were run on a server with 24 X5690 cores at 3.47GHz running under CentOS. For Madagascar we tested system *Mp* (version 0.999) which uses the planning specific variable selection heuristic, and seems to yield the best overall results, according to [102]. Both LAMA and Madagascar were run with their default parameters values, and a CPU cutoff limit of 3600 seconds. In fact LAMA aborts execution in many problems well before the runtime cutoff, as a result of reaching the memory limit.

Due to the large number of parameters, PSP-H was initially run with tens of different combinations for their values. Not surprisingly, its performance varies considerably with these values. In this section we provide some representative results that demonstrate the capabilities and limitation of the system.

In all experiments with PSP-H, procedure optimize was terminated after 300 CPU seconds. Moreover, every time a new (improved) solution was found, the CPU cutoff limit is reset to 120 seconds. As with the other two systems, PSP-H was also run with a CPU cutoff limit of 3600 seconds.

The first column of Table 10, lists the domains of the experimental evaluation, with the name they are known in the literature. The multi-value variable representation of the domains, as they are discovered by LAMA, have different features that directly affect the constraints that are selected by PSP-H for inclusion in the intermediate relaxation part. For Pipesworld and Satellite the initial analysis of PSP-H removes all variables and therefore there is no intermediate relaxation part for problems in this domain. This is reflected in Table 10 by an empty entry for these domains under the fifth column, that refers to a run of PSP-H that requires a non-empty intermediate relaxation part. Among the other domains there are those with counter variables such as

Domain	Number of Problems	LAMA	Madag	PSP-H R1	PSP-H R2	PSP-H R1/R2
Pipesworld	25	25	22	-	25	25
Satellite	22	22	22	-	22	22
Storage	29	20	29	28*	29	28
Openstacks	30	30	18	18	18	18
Elevators	30	30	30	30	30	30
Transport	19	19	19	18*	18*	18
Visitall	28	28	19	22	21	22
Barman	40	40	28	40	40	40
Total	223	214	187	156	203	203

Table 10: Number of problems solved by the planners within a 3600 seconds CPU limit. The second column lists the number of problems tried in each domain. The fifth and sixth column refer to two runs of PSP-H with different parameter values, while the last column combines their results are explain in the paper. An empty entry indicates that the specific parameter setting does not apply to this domain, and a star marks insolvability of some problems due to high memory usage.

the Elevators and Transport, and others such as Storage and Visitall that contain variable with many values. Recall that for the latter domains PSP-H adds to the intermediate relaxation a relaxed mutex constraints of the form $\sum_{i=1}^{n} x_i \leq k$, where x_i are the values of the selected variables, and k an integer value.

Table 10 summarizes the number of problem solved by each system in the tested domains. The second column lists the number of problems tested in each domain, as it was determined by PSP-H's memory consumption. As noted earlier, problems which exceed the memory capabilities of PSP-H are excluded. The third and fourth column depict the problems solved within the CPU cutoff limit by LAMA and Madagascar respectively. The next two column refer to two different characteristic runs of PSP-H that direct the planner towards generating solutions quickly. The parameter values for PSP-H-R1 are as follows. The action and intermediate relaxation parts were both set to 40%, and the initial length of the full relaxation part (parameter h_init_perc) to 50%. Moreover, parameter $restart\_rate\_array$ was set to $restart\_rate\_array =$

$\langle 0.02, 0.04, \ldots, 0.98 \rangle$ (i.e. frequency of restarts=2%), h_red_rate (rate of shortening of the relaxation part) to 15%, and finally $reduce\_r\_red\_rate$ (the reduction rate of the intermediate relaxation part) to 50%. Recall that this setting is not applicable to `Pipesworld` and `Satellite`.

In `PSP-H-R2`, the intermediate relaxation part was empty, whereas the action and final relaxation part were set 40% and 70% respectively. The rest of the parameters are similar to that of `PSP-H-R1` and their specific values are not of interest. What is important to note is that both runs feature frequent restarts and long relaxation parts, thus making the satisfiability problems easier for the underlying boolean optimizer.

Finally, in all runs reported in this work the weights (i.e. the coefficients w_1, w_2, w_3 of the constrained, the action relaxation and the intermediate relaxation parts respectively) were set to 1, 0 and 0.

The last column of Table 10 combines the two `PSP-H` runs by putting together the results for domains with intermediate relaxation part and those for domains without this part. A comparison of that column for `PSP-H` with the one that correspond to `LAMA`, clearly shows that the heuristic planner outperforms `PSP-H`. In fact, the only domain where `PSP-H` (and `Madagascar`) solves more problems is `Storage`.

On the other hand `PSP-H` seems to be on par with `Madagascar` in `Storage`, `Openstacks` and `Elevators`, and can solve more problems in `Pipesworld`. The advantage of `PSP-H` shows clearly in the `Visitall` and `Barman` domains, and this is not accidental. The plans in these domains are long, and it seems that the techniques employed by `PSP-H` address successfully this problem, at least in certain cases. We note that the longest plan that `PSP-H` was able to synthesize comes from `Visitall` and was 248 steps long.

Table 11 presents results about the quality of the plans that were generated by the systems for the largest 5 problems in each domain. The numbers in parentheses are the sums of the number

Domain	LAMA	Madag	PSP-H R2
Pipesworld	(167)	193 (432)	152 (281)
Satellite	(366)	99 (457)	109 (527)
Storage	-	450 (1339)	194 (725)
Openstacks	(662)	448 (606)	396 (572)
Elevators	(839)	428 (2056)	382 (1629)
Transport	(341)	432 (1161)	220 (666)
Visitall	(448)	535 (535)	447 (447)
Barman	(814)	896 (1184)	561 (780)
Total	(3637)	3031 (6431)	2267 (4902)

Table 11: Sums of the solution length (number of parallel steps) and in parentheses the sums of the number of actions for the largest five problems in each domain. In the last line the total counts for all domains except Storage.

of actions in all the 5 problems, whereas the numbers outside parentheses sum the parallel plan lengths, and hence they are no meaningful for LAMA which generates sequential plans. The entries under column PSP-H-R2 refer to the run that appears with the same name in Table 10, and has been presented above.

The fist conclusion of this comparison is that PSP-H generates better quality plans than Madagascar in all domains except for Satellite. In fact, in some domains such as Transport, the difference in favor of PSP-H is substantial. On the other hand, in some domains with low or no action parallelism, such as Openstacks, Visitall and Barman, PSP-H generates plans that have less actions than those in the solutions found by LAMA. However, in domains with high parallelism LAMA generates plans with considerably fewer actions than PSP-H.

PSP-H is controlled by a set of parameters that can be tuned to direct the system towards better quality solutions. One way to accomplish that is by using shorter fully relaxed parts and longer intermediate part. This yields better results, especially in domains with many inequalities in the intermediate relaxation part. For instance, for a characteristic such run, the sum of the plan lengths for the problems that appear in Table 11 are 182 for `Transport` and 300 for `Elevators`, an improvement of more than 20%.

Finally, we note that although some of the domains listed above are unsafe for `PSP-H`. However, in the experimental evaluation the unsafe mode was not activated. On the other hand, it seems that in a few other domains, the unsafe mode fails to avoid dead-ends, and `PSP-H` can only solve a few problems. The most extreme case is that of the `Pegsol` domain, where it only solves a couple of problems.

6.6 Conclusions

This work contributes a number of ideas towards addressing the issue of scalability of the satisfiability-based approach to planning. Preliminary experiments with a prototype implementation of these ideas showed that the new approach can generate long plans of good quality in challenging domains.

The line of research that is taken in this work can be extended in different directions. One question that arises naturally, is whether there are other forms of constraints, stronger than the mutexes, that can provide a more accurate relaxation. It is also important to investigate ways of tuning `PSP-H`'s parameters automatically, based on information that is derived from the problem domain. Not surprisingly, some initial experiments that are not presented here, showed that, domains which are otherwise difficult for `PSP-H`, become substantially easier with a suitable choice of parameter values.

Finally, we view this work as a first attempt to combine constraint solving with ideas from heuristic search that have developed in planning.

Chapter 7

Conclusions and Future Research

In this chapter we present the summary of the contributions of the dissertation and we discuss directions for future research based on the work that has been done so far.

7.1 Summary of Research Contribution

The contributions of this dissertation are in advancing the SAT-based propositional planning. We introduced `SMP`, a novel way to encode the planning problem to propositional formulas (SAT), that is experimentally found to be a significant improvement against other encodings when used in the planning as satisfiability framework. Moreover, we explain our results analytically, by proving that it achieves more propagation than other encodings. Then we used this encoding in a novel method, planning as pseudo-boolean optimization. Our experimental results with an initial implementation of the method, the `PSP` planner, reveal to be an improvement against the planning as satisfiability framework. We further expand the method of planning as pseudo-boolean optimization in the `PSP-H` planner, which enhances the `PSP` planner, by two powerful techniques, incremental goal achievement and heuristic guidance. Our experiments demonstrate that `PSP-H` planner is a competitive planning system for propositional planning, that can find sub-optimal parallel plans of good quality.

In summary, the contributions and the chapters and publications where they appeared are as follows:

I We compared different encodings of the planning as satisfiability framework wrt to the propagation they achieve in a modern SAT solver. Our investigation is limited to unit propagation (UP), since the vast majority of modern SAT solvers use UP as their constraint propagation mechanism. Based on these theoretical results, we explained some of the differences observed in the performance of various planners based on the planning as satisfiability framework. As one would expect, there is a link between their performance and the (unit) propagation they achieve. Based on these results, we introduce a new encoding, the `SMP` and we formally prove that `SMP` renders londex (on a single DTG) [30] implied binary constraints redundant, in contrast to the other encodings that fail to propagate information 'backwards'. We conducted experiments in a number of domains, and verified that `SMP` offers performance improvements. Finally, we used exhaustive search to find non-redundant implied binary clauses for SAT encoding (`SMP`) in various domains, and showed experimentally that adding them in the SAT theory does not bring substantial gains. We present `SMP` in chapter 4 and in [111].

II We presented a novel method for propositional STRIPS planning, planning as (pseudo-boolean) optimization and a first implementation of the method, called `PSP` system. `PSP` planner, follows the classic solve and expand approach, as most planners in the planning as satisfiability framework. It works in two parts, the optimization part and satisfaction part. In the optimization part, `PSP` seeks to maximize the number of goals that can be achieved for successively extended planning horizons, until a plan is found that achieves a number of goals that is equal or grater than some user supplied value. Then PSP enters the satisfaction part. In the satisfaction part the plan that is found in the optimization part is used to identify a new initial state and hence a new planning problem which is solved by invoking the SATPLAN algorithm (SMP encoding). The concatenation of the two plans is the solution of the input planning problem. Any goals that are attained during the optimization part are added to the problem as intermediate facts (unit clauses) that must be true, in order to prune the search space. Finally, we implemented a user controlled strategy that 'slides' the added goals towards the planning horizon. Our experiments demonstrate that PSP is capable of solving (sub-optimally) planning problems that are are unsolvable by SMP. The solutions found by PSP are of high quality (wrt makespan). Finally at an optional post processing step, PSP can improve the solution by successive calls of the SMP for smaller planning horizons. In this way PSP can find optimal plans with runtimes close to the ones of SMP. We present PSP in chapter 5 and in [112].

III We presented the PSP-H system that addresses the limited scalability of SMP and PSP. PSP-H is built on PSP and hence it shares some similarities with it. First of all, PSP-H shares with PSP the planning as pseudo-boolean optimization perspective and, as PSP, it also works in an optimization and satisfaction part. However it extends PSP with two powerful techniques in the optimization part, incremental goal achievement and heuristic guidance. Incremental goal achievement is used to decompose the planning problem into a series of boolean optimization problems. At each sub-problem the objective is to maximize the number of goals that are achieved for successively extended (by one) planning horizons, until a user-defined number of goals is reached. Starting from the initial state, when a sub-problem is solved in the way explained, the plan of this sub-problem is executed on its initial state in order to identify a new initial state and hence a new sub-problem. In each sub-problem PSP−H seeks to maximize a larger number of goals from the previous sub-problem. As in PSP, when a user-defined number of goals is reached (or surpassed), PSP−H proceeds in the satisfaction part which is the same as in PSP. The solution is the concatenation of all the successive plans of the optimization part with the one of the satisfaction part. The second technique, heuristic guidance, is used as a way to mitigate the greedy behavior of the above decomposition method. Heuristic guidance imposes that from the initial states of all sub-problems of the optimization part, there must be a relaxed plan to the final state of the input problem. Three different relaxation methods are used that are based on ignoring some of the problem constraints, but they differ in their strength. Our experimental evaluation on a number of domains taken from planning competitions, demonstrates that PSP−H can solve challenging problems. We present PSP−H in chapter 6 and in [113].

7.2 Future Research

There are several directions for future research, that can enhance the effectiveness of the ideas presented in this thesis. First, the SMP planning system as other planners in the SATPLAN framework, uses the BLACKBOX module to build the planning graph in order to extract the clauses. As explained in section 5.5 some of the benchmarks in the IPC competitions are too demanding for the BLACKBOX system, both in terms of time and memory. Therefore the module for building the planning graph should either be implemented again, or more easily, it would be possible at a preprocessing step to build the planning graph just once for a large planning horizon, store it in a file and reuse it for the successively larger makespans as explained in section 5.5 of chapter 5. Moreover we would like to investigate a way to soundly keep (some) of the learnt clauses found by the SAT solver for a planning horizon to the next planning horizons. Similar ideas are implemented in MaxPlan [128] system. Our experiments revealed that we cannot expect substantial gains with more binary implied clauses. This however, leaves the possibility of employing constraints of higher arity. For example assume a problem with 9 persons, a number of locations and a car that can carry only 4 persons. Then it is easy to see that the constraint $p_1^l(t+1)+\ldots+p_9^l(t+1)-p_1^l(t)-\ldots-p_9^l(t) \leq 4$ is an implied constraint (where $p_i^l(t)$ denotes that person i is at location l at time t). We would like to investigate ways to extract automatically such constraints from domains and the effect that they may have when added (as equivalent clauses, e.g. with the use of *minisat*+ translator) in `SMP`. Another interesting direction is the enhancement of a SAT solver (e.g `precosat`) with a planning specific heuristic, for example as in the `Madagascar` planner [102, 101], or a planning specific propagation method. For example it would be interesting to apply failed literal propagation only to a subset of the unassigned variables in conjunction with unit propagation. Such a set could be for example only the facts that belong to the same state variable v as a goal variable, or at a state variable v' such that there is an edge (v', v) in the causal graph of the problem. Moreover any variable with a distance (in number of layers) more than a threshold d from the planning horizon could be excluded. SAT is used in non propositional planning as well, for example [92, 93, 82]. We conjecture that the methodology and the theoretical analysis regarding the unit propagation that yielded to the `SMP` encoding can be applied to devise more efficient SAT encodings for non propositional planning.

We consider the `PSP` as the 'predecessor' of the `PSP-H`, since `PSP-H` is built on top of `PSP` planner and is experimentally proved to be more efficient. Therefore it seems more reasonable to focus future research on `PSP-H` rather than `PSP`. However, there is an idea that seems worth investigating: In the current implementation in the optimization part, `PSP` attains the goals that are found, and uses them in a 'slide' strategy to prune the search space. The question that arises is whether it is possible to use more or less information in a slide strategy. Intuitively, the more (less) information should produce plans in less (more) time of worse (better) quality. For example assume that the goals that are sliding are $g_1(t_1), g_2(t_2), g_3(t_3), g_4(t_4), g_5(t_5)$, where t_i is a time point, or equivalently $g_1(t_1)+g_2(t_2)+g_3(t_3)+g_4(t_4)+g_5(t_5) = 5$. An example of sliding less information would be to slide the constraint $g_1(t_1) + g_2(t_2) + g_3(t_3) + g_4(t_4) + g_5(t_5) \geq 2$ instead. Moreover in section 5.5 of chapter 5 we discussed in some detail ideas to improve the implementation of PSP planner.

Since the PSP-H planner is built on top of the PSP planner, all the ideas that were discussed in section 5.5 of chapter 5 that may improve the implementation of PSP planner are applicable for PSP-H planner as well. An interesting question is to investigate ways to make more accurate the relaxation method. Since the relaxation method is based on the absence of constraints, an idea is to add some implied constraints in the relax part that are stronger than the mutexes, having always in mind the trade off between accuracy and efficiency. We performed some initial experiments that revealed that PSP-H's efficiency depends heavily on the initial (user-defined) parameters. Indeed domains which are otherwise difficult for PSP-H, become substantially easier with a suitable choice of parameter values. Therefore we believe that is important to investigate ways of tuning PSP-H's parameters automatically, based on information that is derived from the structure of the problem domain. This is also important because of the very large number of combinations of the values of the parameters of PSP-H. Finally, since we view PSP-H as a first attempt to combine constraint solving with ideas from heuristic search, we want to investigate the idea of closing the gap even more: For example it would be interesting to throw a 'PROBE' [80, 79] from the state s at the end of the constrained part towards the goals. Information that would be extracted from a probe that reaches a goal state can be used in a number of interesting ways, besides the obvious one that is to return the plan that is found. For example assume that the length from s to the goal state found by the probe is l. Since the plan found by the probe is sequential, it would make sense to invoke SMP with s as an initial state for successively smaller planning horizons starting from $l - 1$ in order to try to find an optimal plan from s to the goal state, improving the quality of the plan found.

Finally recall that SMP, PSP and PSP-H planners use the SAT solver precosat[16] as a black box, by just invoking its executable. Any advances in the SAT industry, such as a new more efficient SAT solver, can be directly imported into our planners and enhance their performance. Moreover any possible future developments in the modelling of planning as a SAT problem can be directly imported into PSP and PSP-H planners in order to enhance their performance.

Bibliography

[1] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on sat-solvers. In *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, pages 411–425, 2000.

[2] A. Aggoun, Y. Gloner, and H. Simonis. Global constraints for scheduling in chip. In *Invited Industrial Presentation, JFPLC 99*, 1999.

[3] Eyal Amir and Barbara Engelhardt. Factored planning. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 929–935, 2003.

[4] Eyal Amir and Sheila A. McIlraith. Partition-based logical reasoning for first-order and propositional theories. *Artificial Intelligence*, 162(1-2):49–88, 2005.

[5] Alessandro Armando, Claudio Castellini, and Enrico Giunchiglia. Sat-based procedures for temporal reasoning. In Biundo and Fox [19], pages 97–108.

[6] Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *AAAI*, pages 613–619, 2002.

[7] Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *AAAI/IAAI-02, Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada. AAAI Press, 2002*, pages 613–619, 2002.

[8] Rolf Backofen and Sebastian Will. A constraint-based approach to structure prediction for simplified protein models that outperforms other existing methods. In *Logic Programming, 19th International Conference, ICLP 2003, Mumbai, India, December 9-13, 2003, Proceedings*, pages 49–71, 2003.

[9] Christer Bäckström. Computational complexity of reasoning about plans. *Ph.D. thesis, Linkö"ping University, Linkö"ping, Sweden*, 1992.

[10] Christer Bäckström. Equivalence and tractability results for sas+ planning. In *KR-92, Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92). Cambridge, Massachusetts, October 25-29, 1992. Morgan Kaufmann, 1992*, pages 126–137, 1992.

[11] Christer Bäckström and Inger Klein. Planning in polynomial time: the sas-pubs class. *Computational Intelligence*, 7:181–197, 1991.

[12] Christer Bäckström and Bernhard Nebel. Complexity results for sas+ planning. *Computational Intelligence*, 11:625–656, 1995.

[13] Hachemi Bennaceur. The satisfiability problem regarded as a constraint satisfaction problem. In *ECAI*, pages 155–159, 1996.

[14] Christian Bessière and Romuald Debruyne. Theoretical analysis of singleton arc consistency. In *Proceedings ECAI'04 workshop on Modelling and Solving Problems with Constraints, Valencia, Spain*, pages 20–29, 2004.

[15] Christian Bessière and Romuald Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, pages 54–59, 2005.

[16] Armin Biere. P{re,ic}oSAT@SC'09. In *SAT Competition 2009*, page http://fmv.jku.at/precosat/, 2009.

[17] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, pages 193–207, 1999.

[18] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[19] Susanne Biundo and Maria Fox, editors. *Recent Advances in AI Planning, 5th European Conference on Planning, ECP'99, Durham, UK, September 8-10, 1999, Proceedings*, volume 1809 of *Lecture Notes in Computer Science*. Springer, 2000.

[20] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[21] Blai Bonet and Hector Geffner. Planning as heuristic search: New results. In Biundo and Fox [19], pages 360–372.

[22] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33, 2001.

[23] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In de Mántaras and Saitta [33], pages 146–150.

[24] Ronen I. Brafman and Carmel Domshlak. Structure and complexity in planning with unary operators. *J. Artif. Intell. Res. (JAIR)*, 18:315–349, 2003.

[25] Ronen I. Brafman and Carmel Domshlak. Factored planning: How, when, and when not. In *AAAI/IAAI-06, Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA. AAAI Press 2006*, 2006.

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

[27] Yves Caseau and François Laburthe. Heuristics for large constrained vehicle routing problems. *J. Heuristics*, 5(3):281–303, 1999.

[28] P. Chan, K. Heus, and G. Weil. Nurse scheduling with global constraints in chip: Gymnaste. In *In Practical Applications of Constraint Technology (PACT) 1998, London, UK*, March 1998.

[29] Yixin Chen, Ruoyun Huang, Zhao Xing, and Weixiong Zhang. Long-distance mutual exclusion for planning. *Artif. Intell.*, 173(2), 2009.

[30] Yixin Chen, Zhao Xing, and Weixiong Zhang. Long-distance mutual exclusion for propositional planning. In *IJCAI-07, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 1840–1845, 2007.

[31] Yixin Chen, Zhao Xing, and Weixiong Zhang. Long-distance mutual exclusion for propositional planning. In *IJCAI*, pages 1840–1845, 2007.

[32] David W. Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga P. Rajan. Embedded software verification using symbolic execution and uninterpreted functions. *International Journal of Parallel Programming*, 34(1):61–91, 2006.

[33] Ramon López de Mántaras and Lorenza Saitta, editors. *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*. IOS Press, 2004.

[34] Romuald Debruyne and Christian Bessière. Domain filtering consistencies. *J. Artif. Intell. Res. (JAIR)*, 14:205–230, 2001.

[35] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.

[36] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[37] Yannis Dimopoulos and Kostas Stergiou. Propagation in csp and sat. In *CP-06, Twelfth International Conference on Principles and Practice of Constraint Programming September 24-29, 2006 - Cit des Congrs - Nantes, France*, pages 137–151, 2006.

[38] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program.*, 1(3):267–284, 1984.

[39] Stefan Edelkamp and Malte Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In Biundo and Fox [19], pages 135–147.

[40] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT 2003*, pages 502–518, 2003.

[41] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

[42] Michael D. Ernst, Todd D. Millstein, and Daniel S. Weld. Automatic sat-compilation of planning problems. In *IJCAI-97, Fifteenth International Joint Conference on Artificial Intelligence, NAGOYA, Aichi, Japan, August 23-29, 1997*, pages 1169–1177, 1997.

[43] J. W. Freeman. Imrpovements to propositional satisfiability search algorithms. *Ph.D. thesis*, 1995.

[44] Hector Geffner. Planning graphs and knowledge compilation. In Zilberstein et al. [130], pages 52–62.

[45] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013.

[46] Alfonso Gerevini and Ivan Serina. Fast planning through greedy action graphs. In *AAAI-99, Proceedings of the Sixteenth National Conference on Artificial Intelligence, Orlando, Florida, USA, July 1822, 1998*, pages 503–510, 1999.

[47] Malik Ghallab, Dana Nau, and PaoloTraverso. *AUTOMATED PLANNING. theory and practice*. ELSEVIER MORGAN KAUFMANN, 2004.

[48] Fausto Giunchiglia and Roberto Sebastiani. Building decision procedures for modal logics from propositional decision procedure - the case study of modal K. In *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*, pages 583–597, 1996.

[49] Patrik Haslum and Hector Geffner. Admissible heuristics for optimal planning. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14-17, 2000*, pages 140–149, 2000.

[50] Patrik Haslum and Hector Geffner. Heuristic planning with time and resources. In *Proc. ECP-01*, pages 121–132, 2001.

[51] Malte Helmert. A planning heuristic based on causal graph analysis. In Zilberstein et al. [130], pages 161–170.

[52] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res. (JAIR)*, 26:191–246, 2006.

[53] Malte Helmert. *Understanding Planning Tasks: Domain Complexity and Heuristic Decomposition*. Springer-Verlag, Berlin, Heidelberg, 2008.

[54] Malte Helmert. Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.*, 173(5-6):503–535, 2009.

[55] Jörg Hoffmann. Ff: The fast-forward planning system. *AI Magazine*, 22(3):57–62, 2001.

[56] Jörg Hoffmann. Analyzing search topology without running any search: On the connection between causal graphs and h+. *Journal of Artificial Intelligence Research*, pages 155–229, 2011.

[57] Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *JAIR, Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[58] Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *J. Artif. Intell. Res. (JAIR)*, 22:215–278, 2004.

[59] John Hooker. *Logic-Based Methods for Optimization.* Wiley - Intersciense Series in Discrete Mathematics and Optimization, 2000.

[60] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. A novel transition based encoding scheme for planning as satisfiability. In Maria Fox and David Poole, editors, *AAAI*. AAAI Press, 2010.

[61] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. SAS+ planning as satisfiability. *J. Artif. Int. Res.*, 43(1):293–328, 2012.

[62] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. Sas+ planning as satisfiability. *CoRR*, abs/1401.4598, 2014.

[63] Franjo Ivancic, Ilya Shlyakhter, Aarti Gupta, and Malay K. Ganai. Model checking C programs using F-SOFT. In *23rd International Conference on Computer Design (ICCD 2005), 2-5 October 2005, San Jose, CA, USA*, pages 297–308, 2005.

[64] Peter Jonsson and Christer Bäckström. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100(1-2):125–176, 1998.

[65] Subbarao Kambhampati. Planning graph as a (dynamic) csp: Exploiting ebl, ddb and other csp search techniques in graphplan. *JAIR, Journal of Artificial Intelligence Research*, 12:1–34, 2000.

[66] Erez Karpas and Carmel Domshlak. Cost-optimal planning with landmarks. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 1728–1733, 2009.

[67] George Katsirelos and Fahiem Bacchus. Gac on conjunctions of constraints. In *CP-01, Seventh International Conference on Principles and Practice of Constraint Programming, Nov 26 - Dec 1, 2001 Coral Beach Hotel and Resort, Paphos, Cyprus*, pages 610–614, 2001.

[68] George Katsirelos and Fahiem Bacchus. Generalized nogoods in csps. In *AAAI/IAAI-05, Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 390–396, 2005.

[69] Michael Katz and Jörg Hoffmann. Red-black relaxed plan heuristics reloaded. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search, SOCS 2013, Leavenworth, Washington, USA, July 11-13, 2013.*, pages 489–495, 2013.

[70] Michael Katz, Jörg Hoffmann, and Carmel Domshlak. Who said we need to relax all variables? In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*, pages 126–134, 2013.

[71] Henry A. Kautz, David A. McAllester, and Bart Selman. Encoding plans in propositional logic. In *KR-96, Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996. Morgan Kaufmann, 1996*, pages 374–384, 1996.

[72] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI-92, 10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings. John Wiley and Sons, Chichester, 1992*, pages 359–363, 1992.

[73] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *AAAI-96, Proceedings, The Thirteenth National Conference on Artificial Intelligence, August 48,, 2006, Portland, Oregon, USA. AAAI Press 1996*, pages 1194–1201, 1996.

[74] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *AAAI*, pages 1194–1201, 1996.

[75] Henry A. Kautz and Bart Selman. Unifying sat-based and graph-based planning. In *IJCAI*, pages 318–325, 1999.

[76] Henry A. Kautz, Bart Selman, and Joerg Hoffmann. SATPLAN: Planning as satisfiability. In *Booklet of the 5th Planning Competition*, 2006.

[77] Elena Kelareva, Olivier Buffet, Jinbo Huang, and Sylvie Thiébaux. Factored planning using decomposition trees. In *IJCAI*, pages 1942–1947, 2007.

[78] Wei Li and Peter van Beek. Guiding real-world sat solving with dynamic hypergraph separator decomposition. In *ICTAI-04, 16th IEEE International Conference on. Tools with Artificial Intelligence. ICTAI 2004. 15-17 November 2004. Boca Raton, Florida*, pages 542–548, 2004.

[79] Nir Lipovetzky. Structure and inference in classical planning. *Ph.D. thesis, Universitat Pompeu Fabra, Barcelona Spain*, 2012.

[80] Nir Lipovetzky and Hector Geffner. Searching for plans with carefully designed probes. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling, ICAPS 2011, Freiburg, Germany June 11-16, 2011*, 2011.

[81] Adriana Lopez and Fahiem Bacchus. Generalizing graphplan by formulating planning as a csp. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 954–960, 2003.

[82] Qiang Lu, Ruoyun Huang, Yixin Chen, You Xu, Weixiong Zhang, and Guoliang Chen. A sat-based approach to cost-sensitive temporally expressive planning. *ACM TIST*, 5(1):18, 2013.

[83] David G. Luenberger and Yinyu Ye. *Linear and Nonlinear Programming Third Edition*. Springer, 2008.

[84] Inês Lynce and João P. Marques Silva. An overview of backtrack search satisfiability algorithms. *Annals of Mathematics and Artificial Intelligence*, 37(3):307–326, 2003.

[85] Vasco M. Manquinho and João P. Marques Silva. Effective lower bounding techniques for pseudo-boolean optimization. In *DATE*, pages 660–665, 2005.

[86] David G. Mitchell. A sat solver primer. *Bulletin of the EATCS*, 85:112–132, 2005.

[87] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC-01, Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001. ACM 2001*, pages 530–535, 2001.

[88] Christos Papadimitriou. *Computational Complexity*. Addison Welsey Longman, 1994.

[89] J. Porteous and S. (2002) Cresswell. Extending landmarks analysis to reason about resources and repetition. In *Proceedings of the 21st Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG '02)*, pages 42–45, 2002.

[90] Julie Porteous, Laura Sebastia, and Jorg Hoffmann. On the extraction, ordering, and usage of landmarks in planning. *6th European Conference on Planning*, 2001.

[91] Jean-Francois Puget. A fast algorithm for the bound consistency of alldiff constraints. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, pages 359–366, 1998.

[92] Masood Feyzbakhsh Rankooh and Gholamreza Ghassem-Sani. New encoding methods for sat-based temporal planning. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*, 2013.

[93] Masood Feyzbakhsh Rankooh, Ali Mahjoob, and Gholamreza Ghassem-Sani. Using satisfiability for non-optimal temporal planning. In *Logics in Artificial Intelligence - 13th European Conference, JELIA 2012, Toulouse, France, September 26-28, 2012. Proceedings*, pages 176–188, 2012.

[94] Silvia Richter. Landmark-based heuristics and search control for automated planning (extended abstract). In Francesca Rossi, editor, *IJCAI*. IJCAI/AAAI, 2013.

[95] Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 975–982. AAAI Press, 2008.

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

[97] Jussi Rintanen. Evaluation strategies for planning as satisfiability. In de Mántaras and Saitta [33], pages 682–687.

[98] Jussi Rintanen. Compact representation of sets of binary constraints. In *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, pages 143–147, 2006.

[99] Jussi Rintanen. Planning graphs and propositional clause-learning. In *KR*, pages 535–543, 2008.

[100] Jussi Rintanen. Regression for classical and nondeterministic planning. In Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris, editors, *ECAI*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pages 568–572. IOS Press, 2008.

[101] Jussi Rintanen. Heuristics for planning with sat. In David Cohen, editor, *CP*, volume 6308 of *Lecture Notes in Computer Science*, pages 414–428. Springer, 2010.

[102] Jussi Rintanen. Planning as satisfiability: Heuristics. *Artif. Intell.*, 193:45–86, 2012.

[103] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artif. Intell.*, 170(12-13):1031–1080, 2006.

[104] Nathan Robinson. Advancing planning-as-satisfiability. *Ph.D. thesis, Griffith University, Queensland Australia*, 2012.

[105] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. A compact and efficient SAT encoding for planning. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*, pages 296–303, 2008.

[106] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. SAT-based parallel planning using a split representation of actions. In *ICAPS*, 2009.

[107] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. ELSEVIER, 2006.

[108] Stuart Russel and Peter Norvig. *Artificial Intelligence. A Modern Approach*. Prentice Hall Series in Artificial Intelligence, 1995.

[109] L. Ryan. Efficient algorithms for clause-learning sat solvers. In *M.Sc. Thesis, Simon Fraser University*, 2003.

[110] Hossein M. Sheini and Karem A. Sakallah. Pueblo: A modern pseudo-boolean sat solver. In *DATE*, pages 684–685, 2005.

[111] Andreas Sideris and Yannis Dimopoulos. Constraint propagation in propositional planning. In *ICAPS*, pages 153–160, 2010.

[112] Andreas Sideris and Yannis Dimopoulos. Propositional planning as optimization. In *ECAI*, pages 732–737, 2012.

[113] Andreas Sideris and Yannis Dimopoulos. Heuristic guided optimization for propositional planning. In *KR*, pages 669–672, 2014.

[114] Helmut Simonis. Building industrial applications with constraint programming. In *Constraints in Computational Logics: Theory and Applications, International Summer School, CCL'99 Gif-sur-Yvette, France, September 5-8, 1999, Revised Lectures*, pages 271–309, 1999.

[115] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Detection of inconsistencies in complex product configuration data using extended propositional sat-checking. In *Proceedings of the Fourteenth International Florida Artificial Intelligence Research Society Conference, May 21-23, 2001, Key West, Florida, USA*, pages 645–649, 2001.

[116] David E. Smith. Choosing objectives in over-subscription planning. In Zilberstein et al. [130], pages 393–401.

[117] Richard S. Stansbury and Arvin Agah. A robot decision making framework using constraint programming. *Artif. Intell. Rev.*, 38(1):67–83, 2012.

[118] G. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constr. Math. and Math. Logic*, 1968.

[119] A. van Gelder and Y. Tsuji. Satisfiability testing with more reasoning and less guessing. In *Cliques, Coloring and Satisfiability*, pages 559–586, 1996.

[120] Vincent Vidal and Hector Geffner. Solving simple planning problems with more inference and no search. In *CP-05, International Conference onPrinciples and Practice of Constraint Programming, October 1 - 5, 2005, Sitges, Barcelona, Spain*, pages 682–696, 2005.

[121] Vincent Vidal and Hector Geffner. Branching and pruning: An optimal temporal pocl planner based on constraint programming. *Artificial Intelligence*, 170(3):298–335, 2006.

[122] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996.

[123] Toby Walsh. Sat v csp. In *CP*, pages 441–456, 2000.

[124] Laurence A. Wolsey. *Integer Programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization, 1998.

[125] Laurence A. Wolsey and George L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience Series in Discrete Mathematics and Optimization, 1999.

[126] H Zhang. Specifying latin squares in propositional logic. *MIT Press*, 1997.

[127] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

[128] Yixin Chen Zhao Xing and Weixiong Zhang. Maxplan: Optimal planning by decomposed satisfiability and backward reduction. In *ICAPS-06, Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems, June 6-10, 2006, Cumbria, UK*, pages 53–56, 2006.

[129] Lin Zhu and Robert Givan. Landmark extraction via planning graph propagation. In *IN ICAPS DOCTORAL CONSORTIUM*, 2003.

[130] Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors. *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*. AAAI, 2004.