Washington University in St. Louis

Washington University Open Scholarship

All Theses and Dissertations (ETDs)

1-1-2011

Efficient Automated Planning with New Formulations

Ruoyun Huang
Washington University in St. Louis

Follow this and additional works at: https://openscholarship.wustl.edu/etd

Recommended Citation

Huang, Ruoyun, "Efficient Automated Planning with New Formulations" (2011). *All Theses and Dissertations (ETDs)*. 588.
https://openscholarship.wustl.edu/etd/588

This Dissertation is brought to you for free and open access by Washington University Open Scholarship. It has been accepted for inclusion in All Theses and Dissertations (ETDs) by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Yixin Chen, Chair
Christopher Gill
Katz Norman
Yinjie Tang
Kilian Weinberger
Weixiong Zhang

Efficient Automated Planning with New Formulations

by

Ruoyun Huang

A dissertation presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2011
Saint Louis, Missouri

ABSTRACT OF THE DISSERTATION

Efficient Automated Planning with New Formulations

by

Ruoyun Huang

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2011

Research Advisors: Professor Yixin Chen and Professor Weixiong Zhang

Problem solving usually strongly relies on how the problem is formulated. This fact also applies to automated planning, a key field in artificial intelligence research. Classical planning used to be dominated by STRIPS formulation, a simple model based on propositional logic. In the recently introduced SAS+ formulation, the multi-valued variables naturally depict certain invariants that are missed in STRIPS, make SAS+ have many favorable features.

Because of its rich structural information SAS+ begins to attract lots of research interest. Existing works, however, are mostly limited to one single thing: to improve heuristic functions. This is in sharp contrast with the abundance of planning models and techniques in the field. On the other hand, although heuristic is a key part for search, its effectiveness is limited. Recent investigations have shown that even if we have almost perfect heuristics, the number of states to visit is still exponential. Therefore, there is a barrier between the nice features of SAS+ and its applications in planning algorithms.

In this dissertation, we have recasted two major planning paradigms: state space search and planning as Satisfiability (SAT), with three major contributions. First, we have utilized SAS+ for a new hierarchical state space search model by taking advantage of the decomposable structure within SAS+. This algorithm can greatly reduce the time complexity for planning. Second, planning as

Satisfiability is a major planning approach, but it is traditionally based on STRIPS. We have developed a new SAS+ based SAT encoding scheme (SASE) for planning. The state space modeled by SASE shows a decomposable structure with certain components independent to others, showing promising structure that STRIPS based encoding does not have. Third, the expressiveness of planning is important for real world scenarios, thus we have also extended the planning as SAT to temporally expressive planning and planning with action costs, two advanced features beyond classical planning. The resulting planner is competitive to state-of-the-art planners, in terms of both quality and performance.

Overall, our work strongly suggests a shifting trend of planning from STRIPS to SAS+, and shows the power of formulating planning problems as Satisfiability. Given the important roles of both classical planning and temporal planning, our work will inspire new developments in other advanced planning problem domains.

Acknowledgments

I would like to thank my advisors, Professor Yixin Chen and Professor Weixiong Zhang, for their guidance during the course of my graduate study. They taught me how to find important subjects, define suitable research problems, solve problems, and present the results in an informative and interesting way. I would like to thank Professors Chris Gill, Katz Norman, Yinjie Tang, and Kilian Weinberger for serving on my Ph.D. committee and for providing many useful comments and suggestions.

I would also like to thank You Xu, Minmin Chen, Qiang Lu, Zhao Xing, and Guobing Zou in our research group for providing insightful comments on the work and for providing a friendly environment for me to work in.

Ruoyun Huang

Washington University in Saint Louis
August 2011

Contents

Abstract . . . . **ii**

Acknowledgments . . . . **iv**

List of Tables . . . . **viii**

List of Figures . . . . **x**

1 Introduction **1**
1.1 STRIPS versus SAS+ Formulation 1
1.2 Contributions and Significance of Research 2
1.2.1 SAS+ in Planning as Search 3
1.2.2 SAS+ and Planning as Satisfiability 4
1.2.3 Temporally Expressive Planning 5
1.3 Outline 6

2 Background **8**
2.1 STRIPS Formulation 8
2.2 SAS+ Formulation 9
2.3 Domain Transition Graph 12
2.4 Planning Techniques 14
2.4.1 Classical Planning as Heuristic Search 14
2.4.2 Classical Planning as Satisfiability 15
2.4.3 SAT Encoding for Classical Planning 16

3 Abstraction State Space Search **18**
3.1 Greedy Incomplete Plan Retrieving Algorithm (GIR) 19
3.1.1 Materializing a Transition 19
3.1.2 Search for Facts 22
3.1.3 Searching for a Valid Transition Path 23
3.2 Experimental Results 23
3.3 Summary 25

4 Long Distance Mutual Exclusions **26**
4.1 Long Distance Mutual Exclusions From Single DTG 27
4.2 Enhanced Londex Constraints From Multiple DTGs 30
4.2.1 Invariant Connectivity Graphs and Trees 33
4.2.2 Algorithm for Generating londex$_m$ 34
4.2.3 Summary of londex$_m$ Computation 40

4.3 Non-Clausal Londex Constraints . . . 41
4.3.1 Londex as Nonclausal Constraints . . . 42
4.3.2 Effects of Nonclausal Londex Constraints . . . 44
4.4 Experimental Results . . . 45
4.5 Summary . . . 47

5 SAS+ Planning as Satisfiability . . . 48
5.1 SASE Encoding Scheme . . . 49
5.1.1 Search Spaces of Encodings . . . 51
5.1.2 A Running Example . . . 52
5.2 Correctness of SAS+ Based Encoding . . . 53
5.2.1 Solution Structures of STRIPS Based Encoding . . . 53
5.2.2 Equivalence of STRIPS and SAS+ Based Encodings . . . 57
5.3 SAT Solving Efficiency on Different Encodings . . . 65
5.3.1 The VSIDS Heuristic in SAT Solving . . . 65
5.3.2 Transition Variables versus Action Variables . . . 66
5.3.3 Branching Frequency of Transition Variables . . . 68
5.3.4 Transition Index and SAT Solving Speedup . . . 69
5.4 Reducing the Encoding Size of SASE . . . 74
5.4.1 Mutual Exclusion Cliques . . . 74
5.4.2 Reducing Subsumed Action Cliques . . . 74
5.4.3 Reducing Action Variables . . . 75
5.5 Experimental Results . . . 77
5.6 Summary . . . 81

6 Temporally Expressive Planning as Satisfiability . . . 84
6.1 Temporally Expressive Planning . . . 85
6.2 A STRIPS Style Encoding Scheme . . . 87
6.3 Temporally Expressive Planning, A SAS+ Perspective . . . 89
6.4 A Transition Based Encoding for Temporal Planning . . . 91
6.5 Experimental Results . . . 92
6.5.1 The Peer-to-Peer Domain . . . 93
6.5.2 The Matchlift Domain . . . 95
6.5.3 The Matchlift-Variant Domain . . . 96
6.5.4 The Driverslogshift Domain . . . 97
6.5.5 Encoding Efficiency . . . 98
6.6 Summary . . . 99

7 Cost Sensitive Temporally Expressive Planning . . . 100
7.1 Cost Sensitive Temporal Planning Task . . . 101
7.2 Solve CSTE by Weighted Partial Max-SAT . . . 102
7.3 A Branch-and-Bound Algorithm . . . 104
7.3.1 Lower Bounding Based on Relaxed Planning . . . 105
7.3.2 Action Cost Based Variable Branching . . . 111
7.4 Experimental Results . . . 113
7.5 Summary . . . 117

8 Conclusions and Future Research **118**
8.1 Future Works 118
8.1.1 Adaptive Search Strategies for Abstractions 119
8.1.2 Other Semantics and Encodings 119
8.1.3 Additional Techniques for Planning as SAT 120
8.1.4 More Understanding of Structures in General SAT Instances 120
8.1.5 SAS+ in More Advanced and Expressive Planning Models 121

References **122**

Vita **131**

List of Tables

1.1 An outline of how classical planning (with different representations), and temporal planning are tackled by heuristic search and planning as satisfiability; both are the most representative one in their planning method family. The cells with grey color indicate our original works. 3

3.1 The overall results of DTG-Plan(DP) and Fast-Downward (FD) on IPC5 domains. The results are shown in terms of: 1) the number of instance solved by each approach and 2) the average time (sec) to solve one instance. Instances solved by both methods are used to calculate 2). 24

4.1 Invariants of the TPP domain. 34
4.2 Comparisons of the average constraint distances for both fact londex and action londex. Column "Count" indicates the number of constraints we can derive in each problem. Columns 'londex$_1$' and 'londex$_m$' give the average constraint distances of londex$_1$ and londex$_m$, respectively. 41

5.1 The h values of transition variables versus action variables in all domains. Column 'N' is the optimal makespan. Column '$\overline{h}$' is the average and Column 'σ' is the standard deviation. Column '$\overline{h}$ of V_{δ}^{p}' and '$\overline{h}$ of V_{o}^{p}' refer to the average h value of transition variables and action variables in V^{p}, while p equals to 1, 2, 5 or 10. '-' means there is no variable in that percentile range. 68
5.2 Statistics of action cliques, before and after the subsumed action cliques are reduced. "count" gives the number of action cliques, and "size" is the average size of the action cliques. 75
5.3 Number of reducible actions in representative instances. Columns 'R$_1$' and 'R$_2$' give the number of action variables reduced, by unary transition reduction and unary difference set reduction, respectively. Column '%' is the percentage of the actions reduced by both methods combined. 76
5.4 Number of instances solved in each domain within 1800 seconds. SP06^c and SASE^c are short for SP06-Crypto and SASE-Crypto. 80
5.5 Detailed results on various of instances. Column 'Time' is the total running time. Columns 'Var', 'Clause', 'Mem' are the number of variables, number of clauses and memory consumption (in Megabytes), respectively, of the largest SAT encoding. 'TLE' is short for memory limit exceeded and a '-' indicates the planner fails to solve the instance. 82
5.6 Detailed results on various of instances on IPC benchmark domains. 83

6.1 Results on the P2P domain. Crikey2, LPG-c and TFD fail to solve any of the instances. 95
6.2 Results on the Matchlift domain. 96

6.3 Results on the Matchlift-Variant (MLR) domain. 97
6.4 Results on the Driverslogshift domain. TFD cannot solve any instance of this domain. 98

7.1 Experimental results in the P2P domain. Column 'P' is the instance ID. Columns 'T', 'H' and 'C' are the solving time, makespan and total action costs of solutions, respectively. 114
7.2 Experimental results on the Matchlift domain. 115
7.3 Experimental results in the Matchlift-Variant domain. 'TLE' means that the solver runs out of the time limit of 3600s and '-' means no solution is found. 116
7.4 Experimental results in the Driverslogshift domain. The result marked with a '*' means that the solution is invalid. 116

List of Figures

2.1 The DTG generated from the state variable x_{T_1} in Example 1. 13

3.1 Initial state of an example and the DTG of block A 21
3.2 Experimental results (running time) on IPC-5 domains. 24
3.3 Experimental results (number of actions) on IPC-5 domains. 25

4.1 DTGs of the example problem and their causal dependencies. 27
4.2 Enhancement of londex distances based on causal dependencies. 30
4.3 The ICG and ICTs in the TPP domain. 34
4.4 Computing minimum causal dependency cost based on shared preconditions. . . . 36
4.5 An example where shared-precondition enhancement fails to enhance the distance from v to w, but the bridge analysis works. 39
4.6 Propagating a Υ-value through a bridge $f \rightsquigarrow g$. 39
4.7 The numbers of londex clauses used by both clausal and nonclausal methods on Storage-15 (Makespan 8). 44
4.8 Number of instances solved by each planner. 46

5.1 Illustration of how the search spaces of two encoding schemes differ from each other. 51
5.2 Comparisons of variable branching frequency (with $k = 1000$) for transition and action variables in solving certain SAT instances in twelve benchmark domains encoded by SASE. Each figure corresponds to an individual run of MiniSAT. The x axis corresponds to all the decision epochs during SAT solving. The y axis denotes the branching frequency (defined in the text) in an epoch of $k = 1000$. 70
5.3 Comparisons of variable branching frequency (with $k = 1000$) for transition and action variables in solving certain SAT instances in nine other benchmark domains encoded by SASE. 71
5.4 The correlation between SAT solving speedup and the transition indexes. 73
5.5 The results of SASE while different reduction methods are turned on or off. 77
5.6 Number of problems solved by each planner, with increasing limits on running time, memory consumption, number of variables and number of clauses. 79

6.1 This figure partially illustrates temporal dependencies of actions for instances in three domains: Trucks, Matchlift and P2P. Each node represents an action. Each edge represents a temporal dependency between two actions. 93
6.2 The Number of instances that PET and SET can solve, with increasing limits on number of variables or clauses. 99

7.1 A relaxed planning graph for a simple example with 4 actions and 7 facts. For simplicity, no-ops are represented by dots and some action nodes in time steps 1 and 2 are ignored. $\mu(a_1, a_2, a_3, a_4) = (10, 10, 15, 5)$. 106

Chapter 1

Introduction

Since one purpose of artificial intelligence is to grasp the computational aspects of intelligence, mimicking how intelligent agents would behave in the dynamic world, then planning is certainly a key capability of such agents. As more practical considerations, planning techniques have been applied to various real world domains: High Speed Manufacturing [32, 112], Storytelling [97], Anti-Air Defense Operations [6], Mobile Robots [22], Biological Network Planning [15], Human Robot Teaming [120], Personal Activity Planning [100], Natural Language Processing [78, 5] and so forth. Although planning has been vastly applied, to solve planning problems is usually still computationally challenging. In order to be more applicable, automated planning calls for further developments.

Classical planning models the dynamic world using a deterministic, static, finite, and fully observable model. It is relatively restricted, since real world problems usually have lots of complicated properties, for instance, uncertainties, numerics and temporal information etc. Having such restrictions, classical planning is still the key field in planning research. The reason is that it always inspires new techniques and algorithms that are also applicable in advanced planning models. Formulation (may also called *representation* in the literature) is the foundation of a planning algorithm, determining the efficiency of problem solving. Therefore, lots of research efforts have been devoted to studying the formulations of classical planning. The contributions of this dissertation mostly focus on recognizing the potentials of SAS+ formulation, the potentials of formulating planning as satisfiability, and exploiting them for more efficient planning algorithms.

1.1 STRIPS versus SAS+ Formulation

STRIPS used to be the most popular formulation in classical planning. Nevertheless, the recently introduced SAS+ formulation [66, 4] begins to attract more interest. SAS+ is similar to STRIPS in the sense that both are of state transition style. The essential difference is that SAS+ is consisted

of multi-valued variables. Those multi-valued variables naturally describes some *invariants*, which are not explicitly recognized by STRIPS. For example, suppose we have one box and N different places to be put the box on, in STRIPS we will have N propositional facts, indicating all the possible places that the box can be located on. The invariant in this case is the box cannot exist in multiple places. In other words, the invariant implies that in any state exactly one of these N propositions can be true. Such information is naturally carried by SAS+, but not by STRIPS.

It is critical to mention that a STRIPS state space can be very large for a typical planning problem. In particular, the number of binary facts ($\mathcal{F}$) in a planning problem is typically in the order of 10^3 to 10^4, and the size of the state space is $\Omega(2^{\mathcal{F}})$, resulting in huge time and memory complexities in the worst case. To the contrary, in SAS+ the number of multi-valued variables is in the order 100, while each variable's domain is of a size from a few to 20. Furthermore, comparing with the traditional STRIPS formalism, the SAS+ formalism provides structures such as Domain Transition Graphs (DTGs) and Causal Graphs (CGs) to capture vital information of domain transitions and causal dependencies [54].

Given the favorable features of SAS+, research that fully exploit SAS+'s potentials is still limited. SAS+ has been used to derive causal graph heuristics [54], landmarks [102] and automatic heuristic function generation strategy [58]. Most these works focus on exploiting SAS+'s structural information to pursue one single thing: *better heuristics*. It has been pointed out, however, that we will not obtain too much further improvements by just seeking for better heuristics [59]. Helmert has shown that even with an almost perfect heuristic function, the number of states to visit is still exponential. As heuristic function is just a small, even though important, part of the heuristic search model, more big ideas are needed to fertilize this field.

Besides those research on heuristic search, there are very few planning methods that utilize SAS+ formulation. The only such research that we are aware of is a mixed integer programming model [14]. The state of the art, is thus in sharp contrast with the abundance of planning approaches; we have so many planning approaches: partial order causal link planning [133], planning as model checking [33], HTN planning [36] and planning as Satisfiability. All these approaches have interesting characteristics. It is thus both interesting and rewarding to see whether SAS+ can benefit these planning approaches.

1.2 Contributions and Significance of Research

This dissertation is consisted of three parts (as summarized in Table 1.1). The first two parts extend SAS+ to two major planning approaches: heuristic search [21] and SAT-based planning [20, 65],

	Classical Planning		Temporal Planning
	STRIPS	SAS+	
Heuristic Search	Most heuristic search planners are based on STRIPS. Some of them use SAS+ just to derive heuristic functions.	A new abstraction method search on SAS+ (See Chapter 3).	Most existing temporal planners fall into this category.
Planning as SAT	All the existing SAT-based planners are STRIPS based.	Stronger mutual exclusions by utilizing SAS+ (Chapter 4); A new SAT formula based on SAS+ (Chapter 5).	SAT-based temporally expressive planning. Two encodings are studied (Chapter 6). This framework is also extended to handle action costs (Chapter 7).

Table 1.1: An outline of how classical planning (with different representations), and temporal planning are tackled by heuristic search and planning as satisfiability; both are the most representative one in their planning method family. The cells with grey color indicate our original works.

respectively. The third part is to extend SAS+ and planning as SAT to temporally expressive planning and action costs, two advanced planning features. In the following, we discuss the state of the art in these fields and briefly introduce our contributions.

1.2.1 SAS+ in Planning as Search

Built upon the STRIPS formulation, the heuristic search model for classical planning has achieved great success in the last decade; and it is probably still the most popular framework. Also, it is in fact the most active field where the studies of SAS+ take place. Although there is intensive research that apply SAS+ to heuristic search, existing works focus mainly on deriving heuristics by using SAS+ for extra information. That is, the essence of these works is based on STRIPS. In addition, the motivation of studying heuristic search is recently criticized: we will not obtain too much further improvements by seeking for better heuristics [59]. To enrich the heuristic search model, alternative methods to take advantage of SAS+ is seriously needed. Helmert suggests several potential directions [59], for example, symmetry detection [40] or domain simplification [52].

Those possible future directions [59] suggested by Helmert are of course not a complete list. We realize that search in an abstraction state space is also a promising alternative. The basic idea of abstraction state space is following: instead of traversing in the original state space, we restrict and manage our search within an abstracted state space, which is a projection from the original state space. In other words, one state in the abstracted state space corresponds to multiple states in the

original state space. As far as we know, earlier work [76] requires intensive domain analysis, and the hierarchy cut is fixed before the problem solving, which is typically not flexible enough. We can naturally get lots of structural information from SAS+, such as Domain Transition Graph (DTG) and Causal Graph (CG), and build up abstractions accordingly without domain knowledge.

The search in a planning problem however cannot be completely decomposed into searching individual DTGs. DTGs may depend on one another due to causal dependencies, which lead to complex orderings among actions. Hence, causal dependencies are indeed the culprit of the difficulty of automated planning. One possible approach is to merge individual DTGs under the constraints of their causal dependencies, while maintaining the overall graph as small as possible so as to make the search efficient. However, an effective DTG merging scheme is difficult to figure out [57]. This certainly calls for a new approach to utilize DTGs and deal with their causal dependencies.

We propose to search in a space of DTGs and CGs rather than a binary-fact space. Based on the DTG structures, one subroutine of our algorithm directly extracts plans from a graph composed of DTGs. We distribute the decision making into several hierarchical levels of search to deal with causal dependencies. At each level, we design heuristics to order branching choices and prune alternatives that are not promising. We show that the direct search of DTGs can work well across a variety of planning domains, showing competitive performance. Our method is an anytime algorithm which extends state space increasingly [21]. The resulting planner has achieved certain amounts of efficiency improvements in a range of benchmark domains.

The proposed method has at least two advantages over the traditional heuristic search algorithms on STRIPS models such as FF [60]. First, unlike the popular relaxed-plan heuristic that ignores delete effects, the DTGs preserve much structural information and help avoid dead-ends. Second, the proposed method introduces a hierarchy of decision-making where heuristics can be accordingly designed for different levels of granularity of search. In contrast to requiring a single good heuristic in planners, the proposed method provides an extensible framework in which heuristics and control rules can be designed, incorporated and improved at multiple levels.

1.2.2 SAS+ and Planning as Satisfiability

Besides heuristic search, planning as satisfiability is another major paradigm for planning. The approaches using this technique compile a planning problem into a sequence of SAT instances, with increasing time horizons [73], also called makespan. Planning as satisfiability has a number of distinct characteristics that make it effective and widely applicable. It makes a good use of the

extensive advancement in SAT solvers. The SAT formulation can be extended to accommodate a variety of complex problems, such as planning with uncertainty [19], numerical planning [61].

To the best of our knowledge, all existing developments over planning as SAT are based on STRIPS. They do not accommodate to the fact that the benefits of using SAS+ has been widely accepted. We show that SAS+ can be used to derive stronger mutual exclusions [20]. This is the first work that takes advantage of SAS+ in the planning as SAT approach. This work uses redundant constraints to boost the problem solving on the original SAT formula. In other words, the essence of this method is still STRIPS based.

To fully exploit the potentials of SAS+, we in addition propose a SAS+ based SAT encoding scheme (SASE) for classical planning. Unlike previous STRIPS based SAT encoding schemes that model actions and facts, SASE directly models *transitions* in the SAS+ formalism. Transitions can be viewed as a high-level abstraction of actions, and there are typically fewer transitions than actions in a planning task. SASE describes two major classes of constraints: first the constraints between transitions and second the constraints that match actions with transitions. To further improve the performance of SASE, we propose a number of techniques to reduce encoding size by recognizing certain structures of actions and transitions in SASE.

We study the search space induced by SASE, and show that it leads to level by level SAT solvings, with strong empirical evidence support. Although it is not exactly in a level by level manner, from a statistical perspective, transitions do have a significantly higher chance to be decided earlier during SAT solving. We provide such studies, explaining why it is the case under the popular VSIDS framework, and propose how to measure the significance of transition variables. Our study reveals that there is strong correlation between transitions' significance and the speedups.

Our results show that SASE is more efficient in terms of both time and memory usage, and solves some large instances that state-of-the-art STRIPS-based SAT planners fail to solve.

1.2.3 Temporally Expressive Planning

An essential quality of a planner is its modeling capability, which has been been a continuing endeavor of planning research. An important development beyond classical planning is temporal planning, which deals with durative actions occurring over extended intervals of time [94]. Particularly, both preconditions and effects of durative actions can be temporally quantified. Several temporal planning algorithms have been developed in the last decade [94, 118, 3, 53, 48, 79, 133, 129, 46, 47, 38].

Despite their successes, these planners in general have two limitations. First, most existing temporal planners cannot deal with temporally expressive problems. Temporal action concurrency is supported in PDDL2.1 [41], but the fact of lacking supporting planners was not noticed by the research community until recently [26]. A planning problem is temporally expressive if all of its solutions require action concurrency, which indicates that one action occurs within the time interval of another action. Most existing temporal planners are temporally simple without requiring action concurrency [26].

Second, most existing temporal planners either attempt to minimize the total duration of the solution plan (i.e. makespan), or just ignore all quality metrics. However, for many applications, it is required to optimize not only the makespan, but also the total action cost [31], which can represent many quantities, such as the cost of resources used, the total money spent, or the total energy consumed.

We extend the planning as SAT method to temporally expressive planning problems. First we have an encoding scheme for temporally expressive planning based on STRIPS, and extend this approach to handle action cost. We further incorporate SAS+, and apply the idea in SASE to temporally expressive problems.

1.3 Outline

This dissertation is organized as follows.

In Chapter 2, we give a brief introduction to the related topics. In particular, we formalize STRIPS and SAS+ formulations, along with additional topics such as domain transition graph, casual graph and mutual exclusions. We also introduce how planning as search and planning as SAT approaches work.

In Chapter 3 we discuss how to conduct search directly in DTGs. The resulting planner called DTG-Plan, is in fact composed of several components. We explain the details of these components, and how they work together as a complete algorithm. In Chapter 4, we discuss our work on improving mutual exclusion for planning as SAT, called long distance mutual exclusion. We also introduce a stronger long distance mutex by exploiting more information from causal dependencies. While long distance mutual exclusion may consume lots of memory, we in addition present a non-clausal method, which does not trigger the constraints until it becomes necessary.

In Chapter 5, we introduce SASE, the new encoding based on SAS+. We analyze its properties by comparing it with the well known planning graph based encoding, and show that they enforce the

same semantics. We study the search space and the problem structure induced by this new encoding, and explain why it leads to better performance in modern SAT solving algorithms.

In Chapter 6, we extend the planning as SAT method to temporally expressive planning. Two encoding schemes are studied: one based on STRIPS and the other based on SAS+. In Chapter 7, we extend the SAT-based temporal planning framework to further handle action cost. This approach results in MinCost SAT formulas, for which we study two methods to solve them. One method is converting the formula and apply existing MaxSAT solvers. The other is a planning specialized branch and bound algorithm.

Chapter 2

Background

In this chapter, we introduce both STRIPS formulation and SAS+ formulation for classical planning. We also briefly introduce the state of art for classical planning.

2.1 STRIPS Formulation

The traditional planning representation STRIPS is defined over binary-valued propositional facts. A *STRIPS planning problem* is a tuple $\Psi = (\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$, where:

- $\mathcal{F}$ is a set of propositional facts;
- $\mathcal{A}$ is a set of actions. Each action $a \in \mathcal{A}$ is a triple $a = (\textit{pre}(a), \textit{add}(a), \textit{del}(a))$, where pre($a$) $\subseteq F$ is the set of preconditions, and add(a) $\subseteq \mathcal{F}$ and del(a) $\subseteq \mathcal{F}$ are the sets of add facts and delete facts, respectively;
- A state $\varphi \subseteq \mathcal{F}$ is a subset of facts that are assumed true. Any fact not in φ is assumed false in this state. $\varphi_{\mathcal{I}} \subseteq \mathcal{F}$ is the initial state, and $\varphi_{\mathcal{G}} \subseteq \mathcal{F}$ is the goal specification.

We define three action sets. We use ADD(f) to denote the set of actions that have f as one of their add effects, meaning ADD(f) $= \{a \mid f \in add(a)\}$. Similarly, two other action sets are DEL(f) $= \{a \mid f \in del(a)\}$ and PRE(f) $= \{a \mid f \in pre(a)\}$.

An action a is applicable to a state φ if $pre(a) \subseteq \varphi$. We use $apply(\varphi, a)$ to denote the state after applying an applicable action a to φ, in which variable assignments are changed into $(\varphi \setminus \textit{del}(a)) \cup \textit{add}(a)$. We also write $apply(s, P)$ to denote the state after applying a set of actions P in parallel, $P \subseteq \mathcal{A}$, to s. A set of actions P is applicable to φ, when 1) each $a \in P$ is applicable to φ, and 2) there does not exist two actions $a_1, a_2 \in P$ such that a_1 and a_2 are mutually exclusive (mutex) [10]. Two actions a and b are mutex at time step t when one of the following three conditions holds:

- *Inconsistent effects*: $del(a) \cap add(b) \neq \emptyset$ or $del(b) \cap add(a) \neq \emptyset$.
- *Interference*: $del(a) \cap pre(b) \neq \emptyset$ or $del(b) \cap pre(a) \neq \emptyset$.
- *Competing needs*: There exists $f_1 \in pre(a)$ and $f_2 \in pre(b)$, such that f_1 and f_2 are mutex at time step $t-1$.

Two facts f_1 and f_2 are mutex at a time step if, for all actions a and b such that $f_1 \in add(a)$, $f_2 \in add(b)$, a and b are mutex at the previous time step. No facts in the first level, namely the initial state, are mutexes. We call this mutex defined on planning graphs as **P-mutex**, in order to distinguish this mutex from another notion of mutex we discuss in the next section.

A fast but incomplete method to detect mutually exclusive facts and actions is first introduced in Graphplan [10] in which a planning graph with multiple proposition levels is built. Starting from the initial state, the action and fact mutexes in one specific proposition level depend on the mutexes in the previous proposition level. Starting with the interference of actions, mutex of facts and actions can be calculated iteratively until a fix point is achieved.

Definition 1 (Parallel Plan). *For a STRIPS planning problem $\Psi = (\mathcal{F}, \mathcal{A}, \varphi_\mathcal{I}, \varphi_\mathcal{G})$, a parallel plan is a sequence $P = \{P_1, P_2, \dots, P_N\}$, where each $P_t \subseteq \mathcal{A}$, $t = 1, 2, \dots, N$, is a set of actions executed at time step t, such that*

$$\varphi_\mathcal{G} \subseteq apply(\dots apply(apply(\varphi_\mathcal{I}, P_1), P_2) \dots P_N).$$

As a special case of parallel solution plan, a **sequential plan** is a parallel plan $P = \{P_1, P_2, \dots, P_N\}$, where every P_i have exactly one action from $\mathcal{A}$.

2.2 SAS+ Formulation

The SAS+ formalism [4] represents a classical planning problem by a set of multi-valued **state variables**. A planning task Π in the SAS+ formalism is defined as a tuple $\Pi = \{\mathcal{X}, \mathcal{O}, s_\mathcal{I}, s_\mathcal{G}\}$, where

- $\mathcal{X} = \{x_1, \cdots, x_N\}$ is a set of *state variables*, each with an associated finite domain $Dom(x_i)$;
- $\mathcal{O}$ is a set of actions and each action $a \in \mathcal{O}$ is a tuple $(pre(a), \mathit{eff}(a))$, where $pre(a)$ and $\mathit{eff}(a)$ are sets of partial state variable assignments in the form of $x_i = v, v \in Dom(x_i)$;

- A state s is a full assignment (a set of assignments that assigns a value to every state variable). If an assignment $(x = f)$ is in s, we can write $s(x) = f$. We denote $\mathcal{S}$ as the set of all states.
- $s_{\mathcal{I}} \in \mathcal{S}$ is the initial state, and $s_{\mathcal{G}}$ is a partial assignment of some state variables that defines the goal. A state $s \in \mathcal{S}$ is a goal state if $s_{\mathcal{G}} \subseteq s$.

We first define transition and its semantics. We build constraints by recognizing that transitions are atomic elements of state transitions. Actions, cast as constraints as well in our case, act as another layer of logic flow over transitions.

Definition 2 (Transition). *For a SAS+ planning task $\Pi = \{\mathcal{X}, \mathcal{O}, s_{\mathcal{I}}, s_{\mathcal{G}}\}$, given a state variable $x \in \mathcal{X}$, a transition is a re-assignment of x from value f to g, $f, g \in Dom(x)$, written as $\delta^x_{f \to g}$, or from an unknown value to g, written as $\delta^x_{* \to g}$. We may also simplify the notation of $\delta^x_{f \to g}$ as δ^x or δ, when there is no confusion.*

Transitions in a SAS+ planning task can be classified into three categories.

- Transitions of the form $\delta^x_{f \to g}$ are called **regular**. A regular transition $\delta^x_{f \to g}$ is applicable to a state s, if and only if $s(x) = f$. Let $s' = apply(s, \delta^x_{f \to g})$ be the state after applying transition δ to state s, we have $s'(x) = g$.
- Transitions of the form $\delta^x_{f \to f}$ are called **prevailing**. A prevailing transition $\delta^x_{f \to f}$ is applicable to a state s if and only if $s(x) = f$, and $apply(s, \delta^x_{f \to f}) = s$.
- Transitions of the form $\delta^x_{* \to g}$ are called **mechanical**. A mechanical transition $\delta^x_{* \to g}$ can be applied to an arbitrary state s, and the result of $apply(s, \delta^x_{* \to g})$ is a state s' with $s'(x) = g$.

A transition is applicable at a state s only for the above three cases.

For each action a, we denote its **transition set** as $M(a)$, which includes: all regular transitions $\delta^x_{f \to g}$ such that $(x = f) \in pre(a)$ and $(x = g) \in \mathit{eff}(a)$, all prevailing transitions $\delta^x_{f \to f}$ such that $(x = f) \in pre(a)$, and all mechanical transitions $\delta^x_{* \to g}$ such that $(x = g) \in \mathit{eff}(a)$. Given a transition δ, we use $A(\delta)$ to denote the set of actions a such that $\delta \in M(a)$. We call $A(\delta)$ the **supporting action set** of δ.

For a state variable x, we introduce $\mathcal{T}(x) = \{\delta^x_{f \to g}\} \cup \{\delta^x_{f \to f}\} \cup \{\delta^x_{* \to g}\}$, for all $f, g \in Dom(x)$, which is the set of all transitions that affect x. We also define $\mathcal{T}$ as the union of $\mathcal{T}(x)$, $\forall x \in \mathcal{X}$. $\mathcal{T}$ is the set of all transitions. We also use $R(x) = \{\delta^x_{f \to f} \mid \forall f, f \in Dom(x)\}$ to denote the set of all prevailing transitions related to x, and R the union of $R(x)$ for all $x \in \mathcal{X}$.

Definition 3 (Transition Mutex). *For a SAS+ planning task, two different transitions δ_1 and δ_2 are mutually exclusive if and only if there exists a state variable $x \in \mathcal{X}$ such that δ_1, $\delta_2 \in \mathcal{T}(x)$, and one of the following holds:*

1. *Neither δ_1 nor δ_2 is a mechanical transition.*
2. *At least one of δ_1 and δ_2 is mechanical transition, and δ_1 and δ_2 transit to different values.*

Two transitions that are both mechanical or not, are mutual exclusive to another, as long as they belong to the same state variable. If exactly one of them is mechanical, then they are mutually exclusive if and only if they transit to different values.

A set of transitions T is applicable to a state s when 1) every transition $\delta \in T$ is applicable to s, and 2) there do not exist two transitions $\delta_1, \delta_2 \in T$ such that δ_1 and δ_2 are mutually exclusive. When T is applicable to s, we write $apply(s, T)$ to denote the state after applying all transitions in T to s in an arbitrary order.

Definition 4 (Transition Plan). *A transition plan is a sequence $\{T_1, T_2, \ldots, T_N\}$, where each T_t, $t \in [1, N]$, is a set of transitions executed at time step t, such that*

$$s_{\mathcal{G}} \subseteq apply(\ldots apply(apply(s_{\mathcal{I}}, T_1), T_2) \ldots T_N).$$

In a SAS+ planning task, for a given state s and an action a, when all variable assignments in $pre(a)$ match the assignments in s, a is *applicable* in state s. We use $apply(s, a)$ to denote the state after applying a to s, in which variable assignments are changed according to $eff(a)$.

Definition 5 (S-Mutex). *For a SAS+ planning task $\Pi = \{\mathcal{X}, \mathcal{O}, s_{\mathcal{I}}, s_{\mathcal{G}}\}$, two actions $a_1, a_2 \in \mathcal{O}$ are S-mutex if and only if either of the following holds:*

1. *There exists a transition δ, $\delta \notin R$, such that $\delta \in M(a_1)$ and $\delta \in M(a_2)$.*
2. *There exist two transitions δ and δ' such that they are mutually exclusive to each other and $\delta \in M(a_1)$ and $\delta' \in M(a_2)$.*

We named the mutex in SAS+ planning S-mutex to distinguish it from the P-mutex defined in STRIPS planning. We will show in Chapter 5 that these two mutual exclusions induce equivalent

semantics. Therefore, we in general use the single term *mutual exclusion (mutex)* for both, unless otherwise indicated.

For a SAS+ planning task, we write $apply(s, P)$ to denote the state after applying a set of actions P, $P \subseteq \mathcal{O}$, to s. A set of actions P is applicable to s when 1) each $a \in P$ is applicable to s, and 2) there are no two actions $a_1, a_2 \in P$ such that a_1 and a_2 are S-mutex.

Definition 6 (Action Plan). *For a SAS+ task, an action plan is a sequence $P = \{P_1, \dots, P_N\}$, where each P_t, $t \in [1, N]$, is a set of actions executed at time step t such that*

$$s_{\mathcal{G}} \subseteq apply(\dots apply(apply(s_{\mathcal{I}}, P_1), P_2) \dots P_N).$$

The definition of an action plan for SAS+ planning is essentially the same as that for STRIPS planning (Definition 1). There always exists a unique transition plan for a valid action plan. In contrast, given a transition plan, there may exist either no or multiple valid action plans. This problem of finding a set of matching actions for a set of given transitions, is in fact an Exact Cover problem [67].

2.3 Domain Transition Graph

To better represent the structural information in SAS+, we further define Domain Transition Graph and Causal Dependency. Both definitions follow Fast-Downward, where they are formalized for the first time [54].

Definition 7 (Domain Transition Graph (DTG)). *Given a SAS+ planning problem $(\mathcal{X}, \mathcal{O}, s_{\mathcal{I}}, s_{\mathcal{G}})$ and variable $x \in \mathcal{X}$, its DTG $G_x = (V, E)$ is a directed graph, where $V = \{v | v \in Dom(x)\}$ and $E = \{A(\delta_{u,v}) \mid u, v \in V\}$.*

Each value in $Dom(x)$ is a node in G. Each directed edge (u, v), is a set of actions that all have transition $\delta_{u,v}$. Given a fact f, we use $DTG(f)$ to denote the DTG that has f. For simplicity, given a f and DTG G, we say $f \in G$, if f is in G's vertex set V. Similarly, we say an action $a \in G$, if a is included in any of G's edges.

To illustrate DTGs, consider a simplified transportation domain similar to Depots domain [127]. It has four types of objects: TRUCK, CITY, HOIST and CRATE. In this domain, trucks travel between

cities with crates loaded or not, and in each CITY, there is a hoist whose primary actions are LOAD and UNLOAD of crates.

Example 1 Consider a problem with four cities (L_1, L_2, L_3, L_4), one truck T_1, and one crate C_1, where cities (L_1, L_2, L_3, L_4) have, respectively, hoists (H_1, H_2, H_3, H_4) installed. There are links between L_1 and L_2, L_2 and L_3, and L_3 and L_4. We use two state variables to formulate it, x_{T_1} and x_{C_1}, denoting the locations of the truck and the cargo, respectively. The variables' assignments are $Dom(x_{\mathrm{T}_1}) = \{\mathrm{L}_1, \mathrm{L}_2, \mathrm{L}_3, \mathrm{L}_4\}$ and $Dom(x_{\mathrm{C}_1}) = \{\mathrm{L}_1, \mathrm{L}_2, \mathrm{L}_3, \mathrm{L}_4, \mathrm{H}_1, \mathrm{H}_2, \mathrm{H}_3, \mathrm{H}_4, \mathrm{T}_1\}$. An example state is $s = (x_{\mathrm{T}_1} = \mathrm{L}_1, x_{\mathrm{C}_1} = \mathrm{T}_1)$.

Each of the two state variables we discussed in Example 1 has a DTG. One of them, which models the location of T_1, is illustrated in Figure 2.1 as G_1. The corresponding facts are the vertices. The actions, which make the multi-value variables alter between values, consist the transition edges. In G_1, it is those 'MOVE' actions that make the edges.

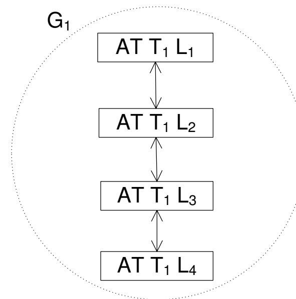


Figure 2.1: The DTG generated from the state variable x_{T_1} in Example 1.

Given two facts f and g, which are both vertices in a DTG G, we compute the shortest path between f and g in G. The length of this shortest path is the minimum number of transitions from f to g in G (Definition 8). Note that in certain planning tasks, two facts may both exist in more than one DTG, we refer DTG cost to the minimum one among all such DTGs, unless otherwise indicated.

Definition 8 (DTG Cost). *Given a domain transition graph G and fact f and g, both in G, the* ***DTG cost*** *from f to g in G, denoted as $\Delta_G(f, g)$, is the length of the shortest path from vertex f to vertex g in G.*

Example 2 For the DTGs in Figure 2.1, let us denote (AT T_1 L_1) as f and (AT T_1 L_4) as g. The DTG cost between these two facts is $\Delta_{G_1}(f, g) = 4$.

Domain transition graphs organize how a state variable transits between different values. While every state variable can be considered as a partial assignment of state, the transitions are essentially partial changes between states. We in addition define causal dependency, which depicts how multiple DTGs, while represented in domain transition graphs, are constrained by each others.

Definition 9 (Causal Dependency). *Given two different DTGs G and G', if there is an action $a \in G'$, such that there is a fact $f \in pre(a)$ and $f \in G$, then G' depends on G, denoted as $G' \in dep(G)$.*

2.4 Planning Techniques

There are three optimization metrics in classical planning: number of actions, total action cost, and makespan. The problem of optimizing the number of action is often called sequential planning, in which case all the actions execute one after another. Actions may in addition have costs then the quality of a plan P is measured by the total action costs of all the actions in P, making the number of action its special case.

The third optimization metric is makespan. By optimizing makespan, we allow multiple actions to be arranged in parallel, as long as there is no mutual exclusion violated [10]. This parallelism semantics is formalized by Graph-Planner [10] for the first time. When discussing planning as SAT, we always assume makespan to be the metric.

As to the state of the art, heuristic search planners currently dominate sequential planning, while SAT based method is much more efficient in optimizing makespan. We briefly introduce these two different planning approaches in the following.

Note that in this dissertation we assume planning problems are based on PDDL [86, 41] planning modeling language. PDDL is practically the standard of all benchmarks in the planning community. In its early versions, PDDL only supports proposition based actions, but nowadays it supports many advanced features.

2.4.1 Classical Planning as Heuristic Search

Planning as heuristic search has achieved great successes in the last decade. The research on planning as search may use two different types of spaces: state space model and plan space model (a.k.a.

partial ordering planning in some literatures [133]). Note in this dissertation when discussing heuristic search, we always refer to the state space model.

Heuristic search has been used in planning for long time. A systematic study on heuristic search planning is HSP [11]. For the first time, a heuristic function automatically derived from the PDDL specification, is used to guide the search. It has started a series of intensive research efforts leading to more powerful systems. The next milestone along this line of research, Fast-Forward [60], uses a better heuristic function, called relaxed planning graph. After that, SAS+ begins to show its power in a practical planning system. A planner called Fast-Downward [54] devises a new heuristic function called *casual graph* heuristic, accompanied by various other techniques. Note that state transiting in Fast-Downward is still STRIPS. Later on, even more heuristic functions are devised [102]. Most these works above generate sub-optimal solutions. It is not until recently that heuristic search is used for optimal sequential planning [58]. SAS+ is adopted by a pattern database heuristic derivation framework, which has achieved significant success over existing optimal sequential planners. Certain limitations still exist though: this heuristic usually heavily relies on parameter tunings.

As mentioned above, there are heavy portion of planning research devoted to heuristic search, especially in *improving the quality of heuristic functions*. These research is however mostly orthogonal to this dissertation. Helmert points out that even if we have an almost perfect heuristic, to conduct search is very expensive. Therefore, the advances in this field call for alternative techniques. In Chapter 3, we will discuss one alternative technique called abstraction search. The idea of abstraction itself is not completely new. The most well known research in this category is the HTN planning [36, 49]. This work has a limitation of being domain dependent. In other words, domain knowledge from human experts are needed to determine the hierarchies in planning tasks. The only recent work uses an idea similar to abstractions, but is based on its own modeling specifications [76].

2.4.2 Classical Planning as Satisfiability

Boolean Satisfiability (SAT) Problem is one of the most important and extensively studied problems in computer science. Given a propositional Boolean formula (Definition 10), SAT problem is to find a variable assignment that satisfies all the clauses, or to prove that no such assignment exists. SAT is the first problem shown to be NP-complete [25, 67].

Definition 10 (Satisfiability Problem). *A SAT instance is a tuple (V, C), where V is a set of variables and C is a set of clauses. Given a SAT instance (V, C), an* ***assignment*** *Γ sets every variable $v \in V$ to be true or false, denoted as $\Gamma(v) = \top$ or $\Gamma(v) = \perp$. If an assignment Γ makes every clause in C to be true, then Γ is a* ***solution*** *to (V, C).*

The standard formula for SAT solvers' input is Conjunctive Normal Form (CNF), which is a Boolean formula involving a set of Boolean variables and a conjunction of a set of disjunctive clauses of literals, with each literal being either a variable or its negation. Most modern SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure. DPLL is a depth-first branch-and-search procedure and often presented as a recursive procedure. SAT algorithms are nowadays very sophisticated, and so efficient that can handle very large problems.

SAT-based planners often use generic SAT solvers as a blackbox. This method has at least two advantages. First, different types of generic SAT solvers can be adopted easily, so that the latest development in SAT research can be always fully utilized. Second, planning-specific constraints can be explicitly encoded in a SAT formula to make the SAT solving efficient.

2.4.3 SAT Encoding for Classical Planning

The overall planning as SAT method is quite straightforward: compile planning problem up to a fixed bound, call a SAT solver to solve the compiled formula, and repeat until a solution is found. Therefore the way that how the compilations work is the key (may also be called encoding scheme in the following).

Encoding scheme could be also considered as a way of formulating planning problem. It converts planning problem to a more general form of problem formulation, in our case SAT formulas. Encoding scheme has great impacts on the efficacy of SAT-based planning. It is pointed out in various literatures that different formulas to the same problem may have huge differences in terms of computational properties [114]. To develop novel and superior SAT encoding schemes has great potentials to advance the state-of-the-art of planning.

Extensive research have been done on devising SAT encoding for planning. The encoding scheme by SatPlan06 [74] (denoted as **PE** in the following) is the most representative one. It enforces the same semantics as the one defined by the planning graph. To facilitate the encoding, SatPlan06 introduces a dummy action dum_f which has f as its both precondition and add-effect. We use $\mathcal{A}^+$ to denote the set of actions when dummy actions are added, which is $A \cup \{dum_f \mid \forall f \in \mathcal{F}\}$. Unless otherwise indicated, action set ADD(f), DEL(f), and PRE(f) all include the corresponding dummy actions.

Let us denote a PE encoding up to time step N as PE(Ψ, N), for a given STRIPS task $\Psi = (\mathcal{F}, \mathcal{A}, \varphi_\mathcal{I}, \varphi_\mathcal{G})$. As a SAT instance, PE($\Psi, N$) is defined by (V, C), where $V = \{W_{f,t} | f \in \mathcal{F}, t \in [1, N+1]\} \cup \{W_{a,t} | a \in \mathcal{A}^+, t \in [1, N]\}$. $W_{f,t} = \top$ indicates that f is true at t, otherwise $W_{f,t} = \perp$. The clause set C includes the following types of clauses:

I. Initial state: $(\forall f, f \in \varphi_{\mathcal{I}})$: $W_{f,1}$;

II. Goal state: $(\forall f, f \in \varphi_{\mathcal{G}})$: $W_{f,N+1}$;

III. Add effect: $(\forall f \in \mathcal{F}, t \in [1, N])$: $W_{f,t+1} \rightarrow \bigvee_{\forall a, f \in add(a)} W_{a,t}$;

IV. Precondition: $(\forall f \in \mathcal{A}^{+}, t \in [1, N]))$: $W_{a,t} \rightarrow W_{f,t}$;

V. Mutex of actions: $(\forall a, b \in \mathcal{A}^{+}, t \in [1, N]$, a and b are mutex): $\overline{W}_{a,t} \vee \overline{W}_{b,t}$;

VI. Mutex of facts: $(\forall f, g \in \mathcal{F}, t \in [1, N+1]$, f and g are mutex) : $\overline{W}_{f,t} \vee \overline{W}_{g,t}$;

Clauses in class I and II enforce that the initial state is true at the first time step, and that the goal facts need to be true at the last time step. Clauses in class III specify that if a fact f is true at time step t, then there is at least one action $a \in \mathcal{A}^{+}$ at time step $t-1$ that has f as an add effect. Clauses of class IV specify that if an action a is true at time t, then all its preconditions are true at time t. Classes V and VI specify mutex between actions and facts, respectively.

Chapter 3

Abstraction State Space Search

The domain transitions and causal dependencies in SAS+ have not been fully exploited. Therefore, the research community is looking for more comprehensive studies and deeper understandings over them. Our work is motivated by the possibility of searching directly in graphs composed of DTGs which are further inter-connected by CGs. The compact structures from SAS+ give rise to smaller spaces. Also, the size of each DTG is small and a direct search on DTGs can be efficient. The resulting algorithm called DTG-Plan, is to mitigate enormous number of expanding states during search.

DTG-Plan uses an incremental abstraction state space search in the higher level. It starts from a very small abstraction state space, expands its abstraction state space, to guarantee the completeness. A key component of DTG-Plan is a Greedy Incomplete plan Retrieving (GIR) algorithm as the subroutine. GIR retrieves a plan quickly to satisfy a set of facts, and is called by DTG-Plan each time when expanding a state.

The motivation of abstraction state space is as follows. Suppose we have a planning instance, the goals are in several distinct DTGs, each has a goal fact. Assume we throw away all the precondition constraints, then it is trivial to achieve the goals: just make those transitions leading to the goals in each corresponding DTGs.

Unfortunately, in reality we can never just simply reduce a problem in this way. But still, what if we have a procedure that is *very likely* to find a sequence of actions making a transition possible in constant time? That is exactly what GIR does. GIR sacrifice the completeness for speed, but in most cases it does find a sub-solution. The nice feature here is: even if the incomplete GIR fails, we do not terminate the algorithm. DTG-Plan expands to a larger scope such that the state missed in the previous iteration will be covered in a new iteration. More specifically, given a SAS+ planning task, the abstraction state space works as follows:

1. We start from a set of DTGs D, such that each DTG in D has a goal fact from $s_{\mathcal{G}}$.

2. We conduct a standard A$*$ heuristic search in the state space determined by D, using the relaxed planning graph heuristic function [60]. The initial state and goal state are as original. The successor function, is to apply one transitions of all DTGs in D. Each transition might be either applicable or not (precondition not satisfied). In the later case, we use GIR to retrieve a plan to repair the broken preconditions, then do the state expanding. If GIR fails, we just suppose this expanding is infeasible (i.e. a dead-end).

3. Once we have explored the whole state space induced by D and find a solution, then return it. If there still time, we extend the scope of D and return to Step 1 for one more iteration. The state space scope increasing strategy is based on the causal dependency. Every time we check all the depended DTGs by all the DTG in D, and incorporate certain number of them according to a progress estimation into D.

4. Repeat Steps 2 and 3, unless we have already incorporated all DTGs into D. That is, D covers the whole original search space. To maintain a global hashing table for all the visited states is crucial, because it can prevent us from visiting one state twice (during two iterations).

3.1 Greedy Incomplete Plan Retrieving Algorithm (GIR)

GIR has three components: search for obtaining set of facts (search_factset() as Algorithm 2), search for a valid transition path (search_paths() as Algorithm 3) and materializing a transition (search_transition() as Algorithm 1). Based on the DTGs, GIR directly extracts plans from a graph composed of DTGs. We distribute the decision making into several hierarchical levels of search to deal with causal dependencies. At each level, we design heuristics to order branching choices and prune non-promising alternatives.

For a DTG G and two facts f and h in G, we define **transition path set** $\mathcal{P}(G, f, h)$ to be the set of all possible paths from f to h in G, with the restriction that each vertex can appear at most once in a path. This path set $\mathcal{P}$ is the major reason that makes GIR incomplete.

3.1.1 Materializing a Transition

We now consider search_transition(), listed in Algorithm 1. It takes as parameters a given state S, a DTG G, a transition δ, and a set of forced preconditions to be explained shortly. In this procedure we choose an action a to materialize the transition δ. Before executing a, its preconditions and forced preconditions must also be true. Therefore, the procedure typically returns a plan consisting of a sequence of actions achieving the preconditions, followed by the action a at the end.

Algorithm 1: search_transition(s, G, δ, forced_pre)

Input: a state s, a DTG G, a transition δ, a set of facts forced_pre
Output: a plan sol that realizes δ from s;

```
1  let A' be the set of actions supporting δ, sorted by cost(o) ;
2  foreach action o in A' do
3      if ∃f, f ∈ del(o), f ∈ protect_list then continue;
4      F = { f | f ∈ pre(o) or f ∈ forced_pre };
5      s' ← s;
6      sol ← search_fact_set(s' , F);
7      if sol is valid then
8          s ← s';
9          apply the effects of o to s;
10         sol = sol + {o};
11         return sol; //upon exit, s is the state after executing sol
12 return no_solution;
```

A transition usually has a number of supporting actions, from which one must be chosen. To choose one action, we sort all supporting actions by a heuristic value in Line 2 of Algorithm 1. Given the transition T in a DTG G, we evaluate each of its supporting actions by estimating the number of steps needed to make all its preconditions true. Formally, given an action o and the current state φ, we estimate the cost to enable o as the total minimum DTG cost of all preconditions of o:

$$cost(o) = \sum_{\forall f \in pre(o)} |\Delta_G(\pi(G, \varphi), f)|.$$

In search_transition(), we sort all supporting actions in an ascending order of their costs. Actions with lower costs are tried earlier because their preconditions are likely to be easier to achieve.

Forced preconditions

We denote the set of forced preconditions as forced_pre(T,p). When materializing a transition T in a path p in a plan, it is beneficial to look ahead at the full path p. A wrong plan for the transition may lead the search into a deadend. Although backtracking can solve this problem, it is more efficient to detect such situations beforehand and avoid deadends.

Consider an example of the Blocksworld domain shown in Figure 3.1. There are three blocks A, B and C. Figure 3.1.i) illustrates the initial state, in which A and B are on the table and C is on B. The goal is "ON A B". Figure 3.1.ii) shows a part of the DTG of block A. Because of the transition edges among the three vertices, they must be true at different time steps.

Suppose we make the first transition (Pickup A TABLE), the state of block A becomes "HOLDING A". To achieve "ON A B", the next action in the path is (Put A B), which has two preconditions. The first precondition is "HOLDING A" which is true. The second precondition is "CLEAR B". Since C is on top of B, we need to remove C from B. But that requires the arm to be free, resulting in "HOLDING A" being invalidated. A deadend is thus encountered.

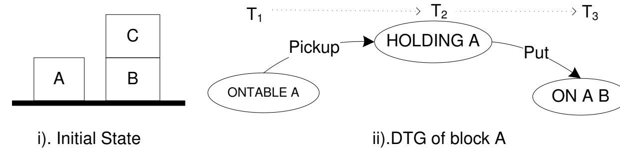


Figure 3.1: Initial state of an example and the DTG of block A

It is clear that the way to avoid this deadend is to achieve "CLEAR B" first before executing (Pickup A TABLE) to make "HOLDING A" true. In general, for each transition T, there may be some facts that need to be made true before an action materializing T is executed. Below we define the forced preconditions, a set of facts that need to be true before transition T is executed.

Given a path p of transitions in a DTG G, and a transition T on path p, we define the forced preconditions of T with respect to p as follows. Assume T turns a fact f to g, and then T_1, which is the next transition following T on path p, changes fact g to h. If there exists a fact q such that $q \in pre(o)$ for any action supporting T_1 and there is a forced ordering $q \prec g$ [77], we call q a forced precondition of transition $T_{f,g}$. The set force_pre(T, p) includes all such forced preconditions of T.

The rationale for the above definition is the following. The fact q is required in order to execute T_1. We need to find a plan to satisfy q when we call search_transition() for T_1. However, in our case, it would be too late due to the $q \prec g$ ordering, which means that g has to be invalidated before achieving q. Thus, we need to achieve q *before* g. In search_transition(), a forced precondition, such as q, will be treated as a precondition for transition T (Line 5).

In the Blocksworld example, "CLEAR B" is a precondition of (Put A B), and there is a forced ordering "CLEAR B" $\prec$ "HOLDING A". Thus, we recognize it as a forced precondition of the transition from "ONTABLE A" to "ON A B". Computing forced preconditions can avoid many deadends during search. It is useful not only for Blocksworld, but also for other domains such as TPP and Rovers.

Algorithm 2: search_fact_set(s, F)

Input: a state s, a set of facts F
Output: a plan sol that achieves facts in F from s;

```
1  if all facts in F are true in s then return {};
2  generate the partial orders O1, O2, and O3;
3  while new valid permutation exists do
4      initialize sol and s' with {} and s, respectively;
5      foreach f, f ∈ F do
6          G ← DTG(f);
7          h ← π(G, s);
8          sol1 ← search_paths(s', G, h, f);
9          if sol1 is valid then break;
10         update s and sol with s' and sol + sol1, respectively;
11     if sol is valid then break;
12 return sol; //upon exit, s is the state after executing sol;
```

3.1.2 Search for Facts

Given a state s and a set of facts F, starting at s, the procedure search_factset() searches for a valid plan that can make all facts $f \in F$ be true at a final state. The facts in F may require a particular order to be made true one by one. The reason of enforcing such a ordering is as follows. The search_factset() procedure allows sub-plans for facts to be interleaved when necessary. It tries to meet the facts ordered earlier whenever possible. Thus when the facts are ordered properly, it is faster to generate plans for each subgoal sequentially with less backtracking. For the facts in F, we have the following partial orders, listed with a descending priority.

O1. For two facts $f, h \in F$, if $DTG(h) \in dep(DTG(f))$, we order f before h. To understand the reason, consider a transportation domain with a cargo and a truck. The DTG of a cargo depends on that of a truck. If we fix the truck at a location, then we may not be able to move the cargo without moving the truck. Thus, it is more reasonable to first deliver the cargo before relocating the truck.

O2. For each fact $f \in F$, we evaluate a degree-of-dependency function defined as

$$\text{degree}(f) = |dep(DTG(f))| - |dep^{-1}(DTG(f))|,$$

where $dep^{-1}(G)$ is the set of graphs that G depends on. For two facts $f, h \in F$, if $\text{degree}(f) > dep(DTG(h))$, we order f before h. This is a generalization of O1.

O3. For two facts $f, h \in F$, if there is a forced ordering $f \prec h$, we order f before h.

Algorithm 3: search_paths(s, G, f_1, f_2)

Input: a state s, a DTG G, facts f_1,f_2
Output: a plan sol;

```
1  if f1 ∈ protect_list then return no_solution;
2  foreach p ∈ P(G, f1, f2) do
3      compute forced_pre(δ, p) for each transition δ in p;
4      sol ← {} ;
5      foreach δ ∈ p do
6          s' ← s;
7          sol ← sol+ search_transition(s' , G , δ, forced_pre(δ, p));
8          if sol is not valid then break;
9          s ← s';
10     if sol is valid then
11         return sol;   //upon exit, s is the state after executing sol;
12 return no_solution;
```

In search_factset(), we first consider all the permutations that honor the above partial orderings. In very rare cases, when all permutations that meet O1, O2 and O3 fail, we continue to compute the remaining orderings.

3.1.3 Searching for a Valid Transition Path

Given a state s, a graph G and two facts $f_1, f_2 \in G$, procedure search_paths() tries to find a valid transition path from f_1 to f_2. First, we compute the set of possible paths $\mathcal{P}(G, f_1, f_2)$. For each p in $\mathcal{P}(G, f_1, f_2)$, we first compute its forced preconditions and then traverse along p to form a plan (Lines 6-11). For each transition δ in p, we call procedure search_transition() to search for a plan. A plan is returned when solutions to all transitions of a path p is found.

3.2 Experimental Results

We test DTG-Plan on the STRIPS domains of the 5th International Planning Competition (IPC5). We compare DTG-Plan with Fast-Downward [54]. Fast-Downward is one of the top STRIPS planners, which uses the DTGs in SAS+ formalism to compute the heuristic. Therefore, we can directly compare a method of directly searching in DTGs (DTG-Plan) against a method using DTGs for deriving a heuristic (Fast-Downward). To carry out the comparison, we compile the latest version

	Instances		Average Time	
	DP	FD	DP	FD
OpenStack	28	28	3.39	10.91
Pathways	30	30	1.69	2.90
Rovers	40	40	12.05	9.69
Storage	17	18	0.58	1.78
TPP	30	30	9.16	47.16
Trucks	16	10	19.19	53.04

Table 3.1: The overall results of DTG-Plan(DP) and Fast-Downward (FD) on IPC5 domains. The results are shown in terms of: 1) the number of instance solved by each approach and 2) the average time (sec) to solve one instance. Instances solved by both methods are used to calculate 2).

of Fast-Downward on our computers, and ran all experiments on a computer with a 2.0MHZ Xeon CPU and 2GB memory and within a 1800 seconds time limit for each run.

Table 3.1 summarizes the results in six out of seven STRIPS domains used in IPC5. Figure 3.2 gives details on some larger instances. For each domain, we show the ten highest numbered instances for which at least one of two methods can solve. DTG-Plan currently does not have competitive performance on the seventh domain, Pipesworld. The reason is that solving this domain requires interleaving of paths not in goal-level DTGs but in other DTGs.

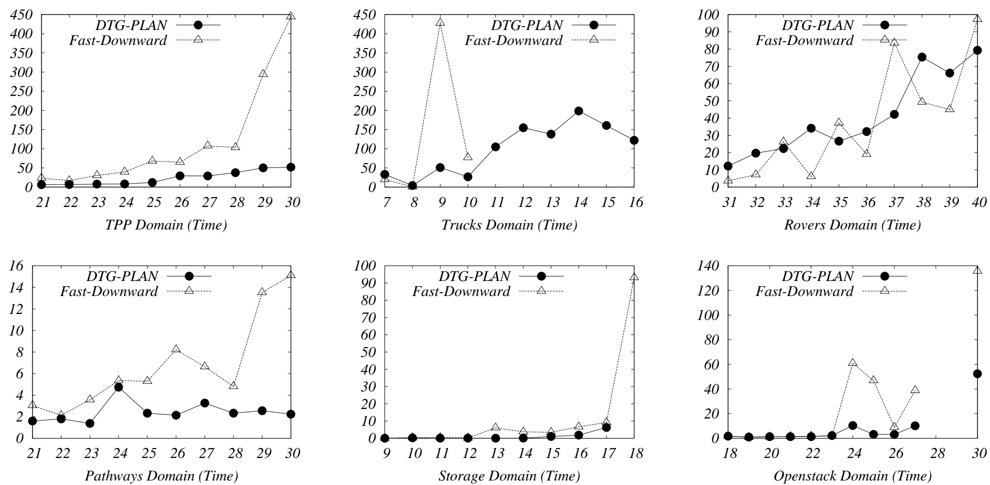


Figure 3.2: Experimental results (running time) on IPC-5 domains.

From the experimental results, DTG-Plan is faster than Fast-Downward on most larger instances, sometimes by more than ten times. For instance, in the Trucks domain, DTG-Plan can solve six large instances that Fast-Downward failed.

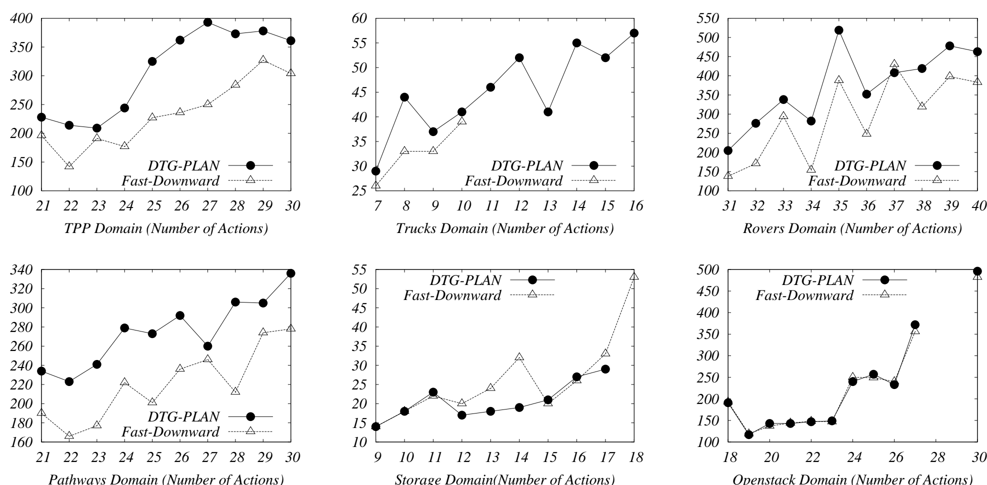


Figure 3.3: Experimental results (number of actions) on IPC-5 domains.

Regarding the number of actions, the two planners differ by less than 20% in most cases. DTG-Plan is better on Storage, while Fast-Downward is better on TPP, Pathways, and Rovers. In the current version of DTG-Plan, we focused on heuristics that improve time efficiency. The fast speed of DTG-Plan allows us to deliver high-quality anytime solutions for a given time limit.

3.3 Summary

DTG-Plan directly searches for plans in a graph composed of DTGs. We distribute the decision making over several search hierarchies to deal with causal dependencies. For each level, we have developed heuristics to order branching choices and to prune nonpromising alternatives. We have experimentally showed that the strategy of direct search in DTGs can work well across a variety of planning domains. Comparison against a leading STRIPS planner has showed that our method is competitive with the state-of-the-art heuristic planner that uses the SAS+ formalism for constructing heuristics. A limitation is that the efficiency of DTG-Plandepends on the problem structure. DTG-Plan works well on IPC-5 domains, but it does not show very clear advantages on IPC-6 domains.

The hierarchy of the proposed search algorithm may seem to be complex, but very often the search can extract a plan quickly along a depth-first path with little backtracking. Since the sizes of DTGs are generally small, the search can be very efficient with proper pruning and ordering heuristics.

The proposed work provides an extensible framework in which strategies can be designed and refined at multiple levels of search, leaving many opportunities for future improvement. Our study is the first attempt towards searching in the DTG space instead of the traditional state space.

Chapter 4

Long Distance Mutual Exclusions

In this chapter, we study the role of constraints in planning as SAT. Planning as SAT usually optimizes makespan, in which case multiple actions may execute in parallel, as long as they are not mutex, as defined in Section 2.1. In fact, sequential planning can be considered as special case of parallel planning, with every individual action to be mutual exclusive to the other.

Mutual exclusions' vital role to parallel planning is not just for correct semantics, but also for better problem solving efficiency. Earlier studies have pointed out that those additional mutex is the key reason to the efficiency of SAT-based planning [74].

We derive Long Distance Mutual Exclusion (londex), a stronger type of mutual exclusion, from the SAS+ formulation. Londex also has two types: fact londex and action londex. Londex constraints specify that certain facts (or actions) cannot be true within some time steps. That is, if we say two facts (or actions) are londex constraints with distance of k, then given a fact (or an action) is true at time t, the other fact can never be true within k time steps away from t. When translated into CNF clauses, londex are also 2-literal clauses. It greatly helps the constraint propagation. Note in this chapter we assume planning tasks are in the STRIPS formulation, while londex is essentially redundant additional information.

We will start with londex$_1$, which is of a simpler form, and then londex$_m$, an enhanced form of constraints. To intuitively illustrate what londex is, let us consider the instance in Example 1 (shown in Figure 4.1). According to londex$_1$, the distance between (LIFTING H_1 C_1) and (LIFTING H_4 C_1) is 2, which is the minimum distance from (LIFTING H_1 C_1) to (LIFTING H_4 C_1) in G_2. This is a lower bound of the time step to obtain the second fact, when given the first one. We can also take causal dependency into account, the lower bound of the distance from (LIFTING H_1 C_1) to (LIFTING H_4 C_1) will be 4 for the following reason. Suppose in the initial state, C_1 is at L_1 and the goal is to move C_1 to L_4. If we only consider the distance in G_2, the minimum distance is 2. However, according to G_1, there must be at least three MOVE actions for T_1 to transit from L_1 to L_4 before H_4 can lift C_1. Hence, at least four time steps are required to reach the goal (LIFTING

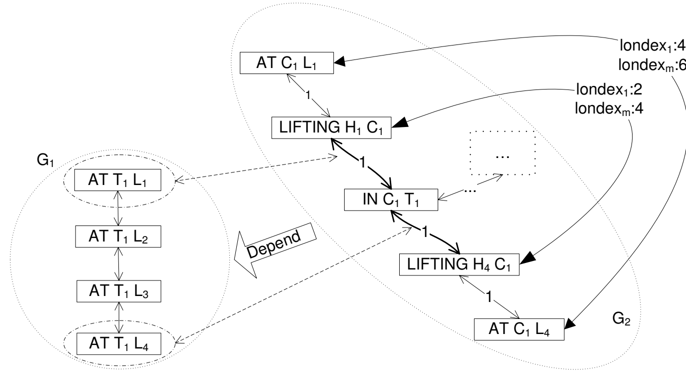


Figure 4.1: DTGs of the example problem and their causal dependencies.

H$_4$ C$_1$) from (LIFTING H$_1$ C$_1$). By considering dependencies among DTGs, tighter bounds can be obtained.

4.1 Long Distance Mutual Exclusions From Single DTG

Broadly speaking, londex can include any constraint that relates actions or facts at different time steps. In the following, we introduce londex$_1$, each of which is derived from only a single DTG.

In order to generate londex$_1$, we first extract fact distance information from a DTG that characterizes the structure of a planning domain. We can generate londex$_1$ from DTG costs. For a given fact f, we use $t(f)$ to denote an arbitrary time step where f is true. This notation also applies to actions.

Definition 11 **(Fact londex)**. Given two facts f_1 and f_2, corresponding to two nodes in a DTG G and $\Delta_G(f_1, f_2) = r$, a fact londex between f_1 and f_2 specifies that: if f_1 and f_2 are true at time steps $t(f_1)$ and $t(f_2)$, respectively, then there exists no valid plan for which $0 \leq t(f_2) - t(f_1) < r$.

Fact londex reflects the minimum distance between the facts belonging to the same DTG. It is possible that f_1 and f_2 can be true at multiple time steps in a plan; all occurrences of f_1 and f_2 must satisfy this constraint.

Example 3 Given the DTGs in Figure 4.1, if (AT C$_1$ L$_1$) is true at time step 0, then (AT C$_1$ L$_4$) cannot be true before step 3.

We now consider londex$_1$ for actions. For simplicity, we say that an action a is *associated* with a fact f if f appears in $pre(a)$, $add(a)$, or $del(a)$. Intuitively, when two facts in a DTG are not too close to each other, two actions associated with the facts cannot be too close to each other either. Without loss of generality, we mark the time steps for actions and facts as follows. For an action a assigned at time step $t(a)$, all of the facts in $pre(a)$ are also true at time step $t(a)$ while all of the facts in $add(a)$ are made true at time step $t(a) + 1$.

We consider two classes of londex$_1$ between two actions a and b.

Class A: Action interference londex. This type of londex specifies that, if actions a and b are associated with a fact f and arranged to be executed at time steps $t(a)$ and $t(b)$, respectively, neither of the following can be true in any valid plan:

$$(1)\ f \in del(a),\ f \in add(b),\ \text{and}\ t(a) = t(b);$$
$$(2)\ f \in del(a),\ f \in pre(b),\ \text{and}\ 0 \leq t(b) - t(a) \leq 1.$$

The above cases (1) and (2) are stronger than the original mutex defined in Section 2.1 because of the inequalities in case (2). If we replace the inequalities in case (2) by $t(a) = t(b)$, cases (1) and (2) are equivalent to the original mutex. To reiterate, a and b may appear more than once in a plan and all multiple occurrences should satisfy these constraints.

Class B: Action distance londex. This type of action londex specifies that, if actions a and b are associated with facts f_1 and f_2, respectively, and it is impossible to have $0 \leq t(f_2) - t(f_1) < r$ following the definition of fact londex, then none of the following can be true:

$$(1)\ f_1 \in add(a),\ f_2 \in add(b),\ \text{and}\ 0 \leq t(b) - t(a) \leq r - 1;$$
$$(2)\ f_1 \in add(a),\ f_2 \in pre(b),\ \text{and}\ 0 \leq t(b) - t(a) \leq r;$$
$$(3)\ f_1 \in pre(a),\ f_2 \in add(b),\ \text{and}\ 0 \leq t(b) - t(a) \leq r - 2;$$
$$(4)\ f_1 \in pre(a),\ f_2 \in pre(b),\ \text{and}\ 0 \leq t(b) - t(a) \leq r - 1.$$

The distance londex are easy to prove. For example, in case (1), if a is executed at time $t(a)$, then f_1 is valid at time $t(a) + 1$. Since the fact distance from f_1 to f_2 is r, f_2 cannot be true until time $t(a) + 1 + r$. Then, since f_2 is an add-effect of b, b cannot be executed until time $t(a) + r$. Other cases can be shown similarly.

Algorithm 4: generate_londex$_1$(Ψ)

Input: A STRIPS planning problem Ψ

Output: londex$_1$ for facts and actions

1 generate the DTGs for Ψ;
2 **foreach** *fact f* **do** $EA1(f) \leftarrow$ generate action interference londex;
3 **foreach** *domain transition graph G* **do**
4   **foreach** *pair of facts* $(f_1, f_2) \in G$ **do**
5   compute $\Delta_G(f_1, f_2)$;
6         $EF1(f_1, f_2) \leftarrow$ generate fact londex ;
7         $EA2(f_1, f_2) \leftarrow$ generate action distance londex ;

Example 4 In Example 1, mutex can only detect the constraints that the truck cannot arrive at and leave the same location at the ***same time***. For example, MOVE(T_1 L_1 L_2) and MOVE(T_1 L_3 L_1) at the same time is mutually exclusive because MOVE(T_1 L_1 L_2) deletes (AT T_1 L_1) while MOVE(T_1 L_3 L_1) adds (AT T_1 L_1). In contrast, londex is stronger as it further specifies that two actions moving the same truck, even if they happen at *different* time points, may conflict with each other. For example, if L_1 and L_4 are 3 steps apart in the DTG, arranging MOVE(T_1 L_1 L_2) at step 0 and MOVE(T_1 L_4 L_3) at step 2 violates a londex but not a mutex.

Note that any londex constraint is *state-independent*, since the cost is a lower bound of fact or action distance in any valid plan regardless of the current state. It is important to emphasize that the londex distance is different from the heuristic function employed by Fast Downward, which is state-dependent. State independencies can be used to compute londex in a preprocessing phase, which can be reused throughout the planning process.

The algorithm for generating londex$_1$ is shown in Algorithm 1, where $EF1(f_1, f_2)$ denotes all fact londex relating facts f_1 and f_2, $EA1(f)$ contains all interference action londex related to a fact f and $EA2(f_1, f_2)$ denotes all action londex related to facts f_1 and f_2.

Londex$_1$ can be generated in polynomial time in the number of facts and the number of actions. Let the number of facts be $|\mathcal{F}|$, the number of actions be $|\mathcal{A}|$ and the number of DTGs be $|G|$. An upper bound of the time complexity of *generate_londex$_1$()* is $O(|G||\mathcal{A}|^2|\mathcal{F}|^2)$. Note that the factor $|\mathcal{A}|$ actually represents the upper bound of the maximum number of actions that have any individual fact as either an add-effect, del-effect or precondition. Empirically, the preprocessing takes less than 30 seconds to generate all londex$_1$ for most of the instances from all IPC competitions [127, 124, 123], which is negligible compared to the planning time that londex$_1$ can help reduce.

Similar to mutex, londex constraints are logically redundant constraints that will not affect the solution space when added or removed from the problem encoding. In general, the more constraints

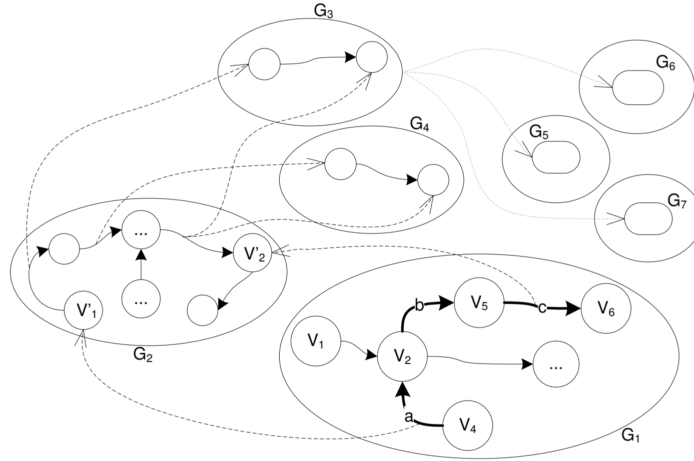


Figure 4.2: Enhancement of londex distances based on causal dependencies.

a planner can detect and utilize, the more pruning power it will have through constraint propagation. The quantity of londex$_1$ constraints is often much larger than that of mutex and thus the former can provide much stronger pruning.

4.2 Enhanced Londex Constraints From Multiple DTGs

Those action and fact londex discussed in the previous section are direct information from DTGs. We further derive stronger londex by recognizing more hidden information from the dependencies between DTGs. Sometimes a transition in a DTG implies a sequence of actions (instead of only one), to enable this transition. Such kind of extended constraint can be detected in certain domains, and leads to further improvements in problem solving.

The idea as illustrated in Figure 4.2, in the high level, is to improve londex by exploiting the causal dependencies (Definition 9). There are seven DTGs, $\{G_1, \dots, G_7\}$, some of which depend on others as indicated by the arrows. In G_1, the path from V_4 to V_6 has the minimum DTG cost of 3 ($V_4 \rightarrow V_2 \rightarrow V_5 \rightarrow V_6$ with transition actions a, b, and c). Suppose a's precondition is V_1', c's precondition is V_2', and V_1' and V_2' are vertices in G_2. Thus, G_1 depends on G_2. Suppose that the minimum DTG cost from V_1' to V_2' is 5 in G_2. The distance between V_4 and V_6 should be at least be $5 + 1 = 6$, a bound tighter than the original value of 3. Moreover, if more dependencies of G_2 exist, we may continue to improve the distance bound by considering more dependent DTGs.

We define unary invariant, which is a restricted form of various invariants that have been studied before. Its definition is based on PDDL modeling language [86, 41], and we assume the planning

instance is specified by PDDL in a concise way of ungrounded actions and predicates. That is, the definitions have templates with parameters of types for objects. Each predicate may have several parameters, each associated with a type of object. Such a compact representation is often expanded to *grounded* facts and actions before planning. Invariants are usually derived from the ungrounded representation.

The predicate grounding operation works as follows. Given a PDDL domain definition, we replace the parameters of each predicate p with objects that have matching types to generate all possible facts. That is, its result, **GROUND**$(p) = \{f_1, f_2, \dots, f_n\}$, is a set of grounded facts. Take predicate $p =$(AT X1$_{TRUCK}$ X2$_{LOCATION}$) as an example. In its definition, X1 is of type *TRUCK* and X2 of type *LOCATION*. Suppose in a given problem, there are two trucks {T$_1$, T$_2$} and two locations {L$_1$, L$_2$}, the grounding operation on p results in GROUND(p)={(AT T$_1$ L$_1$), (AT T$_1$ L$_2$), (AT T$_2$ L$_1$), (AT T$_2$ L$_2$)}.

Definition 12 (Unary invariant). *Given a STRIPS planning domain, an invariant $I = \langle t, P \rangle$ of this domain consists of a set of predicates $P = \{p_1, p_2, \dots, p_n\}$ and a type t such that 1) all predicates in P take an object of type t as a parameter, and 2) among all facts grounded from the predicates in P that have the same instantiation of the parameter of type t, one and only one of these grounded facts can be true in any state.*

There are various types of invariants. The most common type of invariant is represented as a logical expression, indicating there will always be a constant number of facts to be true at any time, for any arbitrary plan P. In this work, we only consider the unary invariants that specify "one and only one fact can be true in a set of facts". This is a restricted form of the invariants thoroughly studied by the AI planning community. It can be considered as a combination of two types of invariants which are called "state membership invariant" and "uniqueness invariant" in TIM [39]. We only use this special class of invariants because it can be used to generate DTGs, based upon which we construct londex. We plan to consider other invariants in our future work.

An invariant generally gives rise to multiple DTGs. Intuitively, a DTG can be viewed as a grounded representation of an invariant. If a DTG G is generated from an invariant I, we write invar(G)=I. We illustrate how to derive DTGs from an invariant using the following two examples. Formal descriptions can be found in the literature [54].

Example 5 In the truck example, an invariant with type TRUCK is:

$$I = \langle \textit{TRUCK}, \{(\text{AT X1}_{\textit{TRUCK}}\ \text{X2}_{\textit{LOCATION}})\} \rangle$$

This invariant implies that a truck can only be at one location at any time. Suppose there are three trucks and three locations, then for each object that is of *TRUCK* type (i.e. T_1, T_2, or T_3), there will be a corresponding DTG. To generate these DTGs, we first plug each of these three objects into the invariant. We will get the following partially grounded formulae, one for each truck: $\{$(AT T_1 X_2)$|\forall$X2$_{LOCATION}$ $\}$, $\{$(AT T_2 X2)$|\forall$X2$_{LOCATION}$ $\}$, and $\{$(AT T_3 X2)$|\forall$X2$_{LOCATION}$ $\}$. Next, for each of these formulae, we get one DTG. For the first formula $\{$(AT T_1 X2)$|\forall$X2$_{LOCATION}$ $\}$, its corresponding DTG indicates that T_1 may be located in different locations. The DTG has three vertices $\{$ (AT T_1 L_1), (AT T_1 L_2), (AT T_1 L_3)$\}$ and its edges can be determined by Definition 7. Similar DTGs can be generated for T_2 and T_3. By doing this, multiple different DTGs are generated from one single invariant.

Example 6 Another invariant, with type CARGO, is:

$$I = \langle \textit{CARGO}, \{(\text{AT X1 X2}), (\text{IN X1 X3})\}\rangle,$$

where X1, X2 and X3 are of types CARGO, LOCATION and TRUCK.

This invariant means that a cargo can either be at a location or in a truck. Multiple DTGs can be generated, one for each cargo. For example, for a cargo C_1, by plugging into the parameter X1$_{CARGO}$ with the concrete object C_1 and perform grounding, we can have a DTG with four vertices: (AT C_1 L_1), (AT C_1 L_2), (AT C_1 L_3), and (IN C_1 T_1).

Definition 13 (Causal Dependency of Invariants). *Invariant I_2 is said to be dependent on invariant I_1, denoted as $I_2 \in dep(I_1)$, if there exist two DTGs, G_1 and G_2, such that $invar(G_1) = I_1$, $invar(G_2) = I_2$, and $G_2 \in dep(G_1)$.*

Example 7 In Figure 4.1, DTGs G_1 and G_2 are from different invariants. The invariant of G_1 is

$$I_1 = \langle \textit{TRUCK}, \{(\text{AT X1 X2})\}\rangle,$$

where X1 and X2 are of type TRUCK and LOCATION, respectively. The invariant of G_2 is

$$I_2 = \langle \textit{CARGO}, \{(\text{AT X1 X2}), (\text{IN X1 X3}), (\text{LIFTING X4 X1})\}\rangle,$$

which has X1 of type CARGO, X2 of type LOCATION, X3 of type TRUCK and X4 of type HOIST. In this example, since G_2 depends on G_1, we also have that I_2 depends on I_1, denoted as $I_2 \in dep(I_1)$.

4.2.1 Invariant Connectivity Graphs and Trees

Corresponding to a dependency graph of DTGs, we can also construct an Invariant Connectivity Graph (ICG), where the nodes correspond to invariants and there is a directed edge from node I_2 to node I_1 if $I_2 \in dep(I_1)$. Each problem instance has one invariant connectivity graph.

Given an ICG, we may choose any invariant I as the root and build a spanning tree of the ICG. This leads to an Invariant Connectivity Tree (ICT) rooted at I.

We have introduced the data structure called causal graph, which is originally introduced in Fast-Downward [54] to represent the dependencies between the DTGs, along with a method for breaking the cycles. An ICG and a causal graph are different. The ICG can be viewed as the ungrounded counterpart of the causal graph. Each vertex in an ICG is an invariant, while each vertex in a causal graph is a DTG. Therefore, the ICG models the dependencies among invariants and the causal graph models the dependencies among DTGs.

The method we use for generating ICTs is different from the cycle-breaking strategy used in the Causal Graph (CG) heuristic [54]. The former simply finds a spanning tree from a certain root node, while the latter orders the nodes by the difference of in-degrees and out-degrees and removes certain edges based on the ordering. In the CG heuristic, the cycle-breaking can be expensive since it is done only once and the CG heuristic uses the same tree to compute the heuristic values for all states. In our algorithm, however, we generate different ICTs for different facts and thus require the cycle-breaking to be fast. Moreover, in our algorithm, we use the invariant under consideration as the root in order to maximize the possible dependencies that we can exploit to enhance londex distances.

We can create different ICTs using different nodes as the root node. Taking the TPP domain used in IPC5 as an example, we can derive five invariants, as shown in Table 4.1. Figure 4.3 illustrates the ICG of the TPP domain and two example ICTs with I_1 and I_3 as the root, respectively.

The purpose of deriving ICTs is to remove cyclic dependencies for computing londex$_m$. Theoretically, any way to break a cyclic dependency is acceptable for the purpose of computing londex$_m$, since londex$_m$ just provide lower bounds on distances. However, when we compute londex$_m$ based on DTGs derived from the same invariant I, we use an ICT with I as its root. This is because we want to take into account as many dependencies as possible, in order to derive tight distance lower bounds. For example, if we use the ICT rooted at I_1 in Figure 4.3 to compute londex$_m$ with respect to invariant I_3, we will only consider I_3's dependencies on I_4 and I_5 while miss its dependencies on I_1 and I_0. Using the ICT rooted at I_3, on the other hand, will include more dependencies.

0: ⟨*TRUCK*, {(AT *TRUCK*,#)}⟩
1: ⟨*GOODS*, {(ON-SALE *GOODS*,#,#)}⟩
2: ⟨*GOODS*, {(STORED *GOODS*,#)} ⟩
3: ⟨*GOODS*, {(READY-TO-LOAD *GOODS*,#,#)} ⟩
4: ⟨*GOODS*, {(LOADED *GOODS*,#,#)} ⟩

Table 4.1: Invariants of the TPP domain.

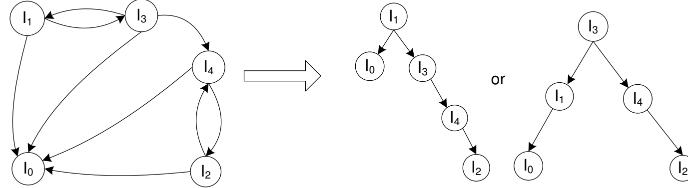


Figure 4.3: The ICG and ICTs in the TPP domain.

Note that there may exist multiple ICTs with the same invariant as the root node. When an invariant I has multiple ICTs with I as the root, we can arbitrarily choose one of these ICTs. Again, any ICT is usable for the purpose of computing londex$_m$ since all we need are lower bounds and hence we can discard some dependencies. For a node I in an ICT Z, we use $dep_Z(I)$ to denote the set of invariants that I depends on within Z.

4.2.2 Algorithm for Generating londex$_m$

Since action londex is derived from fact londex, our strategy is to first enhance the distance in fact londex. After we obtain enhanced fact londex, we use the same definitions used by londex$_1$ to enhance the distances in action londex.

It is not as straightforward as it might seem to augment the fact distance with causal dependencies. The main difficulty is that we need to ensure that the enhanced distance value is a valid lower bound in *any* solution plan, regardless of the initial, goal, or intermediate states. Further, the enhanced distance value must be a lower bound to the distance in parallel plans. Therefore, we need to take the possible parallelization of actions into consideration. In this section, we propose two methods to enhance the distance constraints that satisfy the above requirements. We start with some basic definitions.

Definition 14 (Predecessor and Successor Set). *Given a transition $\delta_{v \to w}$ in a DTG $G(V, E)$, $v, w \in V$, we call w a **successor** of v, and v a **predecessor** of w. For a node u in $G(V, E)$, we*

*define its **successor set** to be* $succ(u) = \{x \mid x \in V, T_{u,x} \in E\}$*, and its **predecessor set** to be* $pred(u) = \{x \mid x \in V, T_{x,u} \in E\}$.

Definition 15 (Shared Precondition). *Given a transition $T_{v,w}$ in a DTG $G(V, E)$, if there is a fact f such that $f \in pre(a)$ for each action $a \in \delta_{v,w}$, we define f as a **shared precondition** for the transition $\delta_{v \to w}$, denoted by $f \mapsto \delta_{v \to w}$. We define $SP(v, w) = \{f | f \mapsto T_{v,w}\}$ to be the **set of shared preconditions** of $\delta_{v \to w}$.*

Notice that the minimum DTG cost $\Delta_G(f_1, f_2)$ is a lower bound of the distance from f_1 to f_2 in G. However, due to shared preconditions and causal dependencies, we may obtain tighter lower bounds than $\Delta_G(f_1, f_2)$. Shared preconditions can be found in most problem instances that we experimentally studied.

Distance enhancement based on shared preconditioning

The idea of augmenting fact londex stems from the observation that some transitions in a DTG may *always* require some transitions in another DTG due to shared preconditions. To be concrete, we illustrate our idea by Figure 4.4. Consider two facts f and g in the same DTG G. When computing the londex$_1$ distance between f and g in G, we use $\Delta_G(f, g)$ as the minimum distance. Consider the shortest path between f and g, $\xi = (f, v_1, \ldots, w_1, g)$. If there is a shared precondition p of $\delta_{f \to v_1}$ and a shared precondition p' of $\delta_{w_1 \to g}$, and if p and p' are also in another DTG G' which G depends upon, we can compute $\Delta_{G'}(p, p')$, the minimum distance between p to p' in G'. If $\Delta_{G'}(p, p') \geq \Delta_G(f, g)$, the minimum cost to transit from f to g through the path ξ can be updated to $\Delta_{G'}(p, p') + 1$ rather than $\Delta_G(f, g)$.

The above enhancement is valid for the following reason. Let P be a parallel plan that transfers f to g by going through v_1 and w_1. Let f and g be true at time t_f and t_g, respectively. Then, since p is the shared precondition for $\delta_{f \to v_1}$, p must be true at some time t_p where $t_p \geq t_f$. Similarly, since p' is the shared precondition for $T_{w_1 \to g}$, p' must be true at some time $t_{p'}$ where

$$t_{p'} < t_g. \tag{4.1}$$

Note that it is impossible to have $t_{p'} = t_g$, because p' is the precondition of the transition $T(w_1, g)$. Therefore, if p' is true at $t_{p'}$, the earliest possible time for g to be true is $t_{p'} + 1$. Therefore, we have $t_g - t_f \geq t_{p'} - t_p + 1$. Hence, $\Delta_{G'}(p, p') + 1$ is a lower bound on the distance between f and g under the condition that the transition goes through v_1 and w_1. Moreover, enumerating all pairs

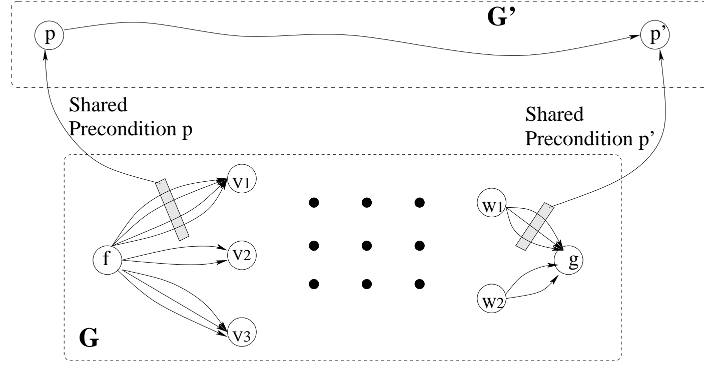


Figure 4.4: Computing minimum causal dependency cost based on shared preconditions.

(v_i, w_i) where $v_i \in succ(f)$ and $w_i \in pred(g)$ will result in a lower bound unconditional of which nodes the transition goes through.

It is important to note that londex helps to compute distance lower bounds in parallel plans instead of sequential plans. Therefore, $\Delta_{G'}(p, p') + \Delta_G(f, g)$ may not be a lower bound, because there may be shared actions between G' and G. Even if G' and G have disjoint action sets, the actions transforming p to p' and the actions transforming f to g may be placed in the same time step in a parallel plan as long as they are not mutually exclusive. $\Delta_{G'}(p, p') + 1$, on the other hand, is a valid lower bound.

The above analysis on shared preconditions provides a mechanism to enhance the distance lower bound between f and g. The distance lower bound between p and p' may also be recursively enhanced through the same dependency analysis. Let $\Upsilon(f, g)$ denote the enhanced distance lower bound between any two facts f and g in the same DTG G, we are interested in computing

$$\Upsilon(f, g) = \min_{v \in succ(f), w \in pred(g)} \left\{ \max \left(\max_{p \in SP(f,v), p' \in SP(w,g)} \{\Upsilon(p, p')\} + 1, \Delta_G(v, w) + 2 \right) \right\} \tag{4.2}$$

However, it is difficult to exactly compute (4.2) because the definition of Υ-value may be cyclic if there are cycles in the dependency graph of DTGs. Fortunately, we are only interested in any tighter lower bound and are not required to have the maximum possible Υ-value. Therefore, in our implementation, for each DTG G, we construct an ICT rooted at $invar(G)$ and consider the shared preconditions p and p' only if they both reside in a DTG G' where $invar(G)$ depends on $invar(G')$ in the ICT. This will effectively remove possible dependency cycles and make (4.2) well defined.

The details of the algorithm for computing the Υ-value is given in procedures Υ-value() and β-value() in Algorithms 2 and 3, respectively. The β-value is similar to the Υ-value, but is intermediate

Algorithm 5: Υ-value(G, b)

Input: A DTG G with its Δ_G values, an integer bound b

Output: $\Upsilon(f, g)$ for every pair $f, g \in G$

```
1  generate an ICT Z with invar(G) as the root;
2  clear the saved β-value for all pairs of facts;
3  foreach pair of facts (f, g), f, g ∈ G do
4  |   if 1 < Δ_G(f, g) ≤ b then Υ(f, g)=β-value(Z,G,f,g) ;
5  |_  else Υ(f, g) = Δ_G(f, g) ;
```

Algorithm 6: β-value(Z,G,f,g)

Input: Invariant connectivity tree Z, DTG G, facts f and g

Output: β-value of facts f,g

```
1  if β-value(f,g) is saved then return the saved value;
2  β = ∞;
3  Generate the ordered succ-pred-pair list L(f, g); // see text for details
4  foreach pair of facts ⟨v, w⟩ in L(f, g) do
5  |   let α be the shortest path of v ⇝ w such that f, g ∉ α;
6  |   if |α| + 2 ≥ β then break;
7  |   m ← |α| + 2;
8  |   foreach ⟨p, p'⟩,p ↦ T(f, v) and p' ↦ T(w, g) do
9  |   |   if p, p' ∈ G' and invar(G') ∈ dep_Z(invar(G)) then
10 |   |_  |_  m ← max(m, β-value(Z, G', p, p')+1);
11 |_  β ← min(m, β) ;
12 save β as β-value(f,g);
13 return β;
```

and temporary. This is because each individual function call of β-*value()* in Υ-*value()* can be based on a different ICT. Since the β-values depend on the ICT generated in Line 1 of Υ-*value()*, all β-values are discarded when a different ICT is used.

The recursive function β-value() is used to retrieve causal dependencies and count them into transition costs, until no further uncalculated information can be found. For each pair of facts (f, g) in a DTG, we enumerate all facts v and w such that $v \in succ(f)$ and $w \in pred(g)$. We order the pairs by their shortest distances. Specifically, in Line 3 of the β-value() algorithm, we generate the *succ-pred-pair list* $L(f, g)$ defined as follows: $L(f, g) = (pair_1, pair_2, \dots, pair_n)$, where $pair_i$ = (v, w), $v \in succ(f)$, and $w \in pred(g)$. $L(f, g)$ is so ordered that the shortest distance between the two facts in $pair_i$ is no greater than the shortest distance between the two facts in $pair_j$, if $i < j$. The purpose of ordering the pairs is to save computation time. When the shortest path from f to g going through $pair_i$ has a length greater than the current enhanced β value, we do not need to consider $pair_i$ or any subsequent pairs in $L(f, g)$ (Line 6).

A special case is when there is a fact v such that $v \in succ(f)$ and $v \in pred(g)$. For such a fact v, we insert the pair (v, v) to the beginning of $L(f, g)$ because the distance of (v, v) is zero.

The distance value can be enhanced if there is a pair of facts (p, p') with the following restriction: a) $p \mapsto \delta_{f \to v}$, b) $p' \mapsto \delta_{w \to g}$, c) both p and p' belong to a DTG G' that does not have the same fact with G, and d) $invar(G)$ depends on $invar(G')$ in the pre-generated ICT Z (Line 9). We can potentially enhance the Υ value when such conditions are met (Line 10). If dependencies over several different pairs of (p, p') can be found, we use the maximum distance lower bound that can be obtained (Line 8–10).

We should note that we save the β-value for every pair of facts under a given ICT (Line 12 of β-value()). During the computation of β-value(Z,G,f,g), we first look up a hash table to see if the β-value(f,g) has been saved already (Line 1 of β-value()). If not, we call β-value(Z,G,f,g) recursively to compute it. When we use a different ICT, we discard all β-values and recompute (Line 2 of Υ-value()).

An integer b is used to parameterize Υ*-value(G, b)*, in which we only try to enhance the distance of (f, g) when $\Delta_G(f, g)$ is greater than one but no more than b. When $\Delta_G(f, g) = 1$, there is a direct transition from f to g and their distance cannot be enhanced through shared preconditions. A larger b leads to potentially longer enhanced distances, while a smaller b requires a smaller computational cost. We find in our experiments that most Υ-value enhancements are obtained when the Δ_G value is 2.

Another technical detail is that a pair of facts may appear in more than one DTG. In this case, their Υ-values will be computed multiple times and we retain the maximum value.

Distance enhancement based on bridge analysis

The Υ-value can enhance the $\Delta_G(f, g)$ value by considering the causal dependencies derived from shared preconditions. However, in some cases, we may not be able to detect shared preconditions by the causal dependency analysis, although there may indeed exist a tighter lower bound. Figure 4.5 shows such an example, where there are five facts $\{v, f, g, h, w\}$ in the DTG. There are shared preconditions p_1 and p_2 satisfying $p_1 \mapsto \delta_{f \to h}$ and $p_2 \mapsto \delta_{h \to g}$. Suppose that the Υ-value of (f, g) is 4 which is greater than the original fact distance $\Delta_G(f, g) = 2$. However, for facts v and w, we cannot find any shared precondition for $\delta_{v \to f}$ or $\delta_{g \to w}$. Therefore, the shared-precondition enhancement cannot be applied to enhance the distance between v and w.

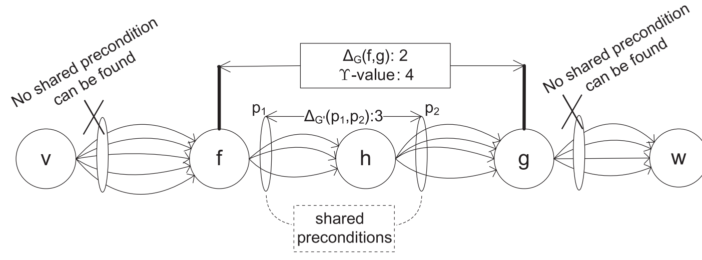


Figure 4.5: An example where shared-precondition enhancement fails to enhance the distance from v to w, but the bridge analysis works.

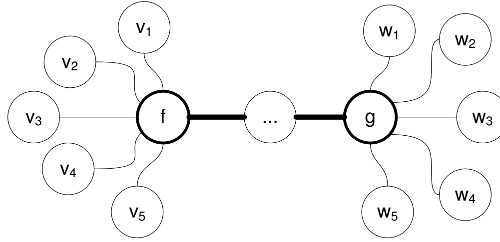


Figure 4.6: Propagating a Υ-value through a bridge $f \rightsquigarrow g$.

Nevertheless, we may still augment the distance value of a pair of facts through what we call *bridge analysis*. For a pair of connected facts v and w in a DTG G, a pair of facts (f, g) is called a **bridge pair** of (v, w) if any path from v to w in G visits f and g in order. A path from f to g is then called a **bridge**.

DTGs are typically sparse and frequently contain bridge pairs. Hence, an enhanced distance value based on a bridge pair can be propagated to other pairs of facts. In Figure 4.6, if the distance of (f, g) is increased to $\Upsilon(f, g)$ by shared preconditions, its improvement can be propagated to all other fact pairs (v_i, w_j) that have (f, g) as a bridge pair. Precisely, the distance of (v_i, w_j) can be improved to:

$$\Delta_G(v_i, f) + \Upsilon(f, g) + \Delta_G(g, w_j). \tag{4.3}$$

The enhanced cost in (4.3) is a lower bound of the distance from v_i to w_j in any parallel plan, because (f,g) is a bridge pair for (v_i, w_j). Any path from v_i to w_j will have the form $v_i \rightsquigarrow f \rightsquigarrow g \rightsquigarrow w_j$, so that the cost of $f \rightsquigarrow g$ will always be part of the cost of $v_i \rightsquigarrow w_j$. Since the three sub-paths $v_i \rightsquigarrow f$, $f \rightsquigarrow g$, and $g \rightsquigarrow w_j$ cannot overlap in any parallel plan, their costs can be added as in (4.3).

In Figure 4.1, the Υ-value of (LIFTING H_1 C_1) to (LIFTING H_4 C_1) is 4. Facts (AT C_1 L_1) and (AT C_1 L_4) have (LIFTING H_1 C_1) and (LIFTING H_4 C_1) as a bridge pair. So the distance from

(AT C_1 L_1) to (AT C_1 L_4) can be improved by Δ_G((AT C_1 L_1), (LIFTING H_1 C_1)) + Υ((LIFTING H_1 C_1), (LIFTING H_4 C_1)) + Δ_G((LIFTING H_4 C_1), (AT C_1 L_4)) = $1+4+1=6$, which is greater than that of 4 as in londex$_1$.

Finally, propagating Υ-value through bridge pairs takes little time because for each DTG, the computation happens within the graph itself and does not require exploration of multiple DTGs in the dependency trees. The propagation allows us to enhance a large number of distances with little cost.

4.2.3 Summary of londex$_m$ Computation

Algorithm 4 summarizes the procedure for generating londex$_m$. It has four steps:

1. (Lines 1–3) Compute londex$_1$. Namely, initialize and compute the minimum DTG costs,
2. (Lines 4–5) Compute the Υ-values for facts whose londex$_1$ distances are no more than 2. We set $b = 2$ because we have found that it is not worth the computational cost to use a larger b. The number of extra distances that can be enhanced using a larger b is very limited and in most cases can also be augmented by the more efficient bridge-pair enhancement in the next step.
3. (Lines 6–10) Perform the bridge-pair enhancement to propagate the Υ-values to other pairs.
4. (Line 11) Generate londex$_m$ for actions. Like londex$_1$, we generate the action londex of londex$_m$ using condition (1)-(4) in Class B of Section 4.1, in which a londex$_1$ fact distance r is replaced by a londex$_m$ fact distance.

The total time complexity of $generate_londex_m()$ is $O(|G||V|^{2d})$, where $|G|$ is the total number of DTGs, $|V|$ the maximum number of vertices in a DTG, and d the maximum depth of any ICT. d is typically a small constant (< 5), and both $|G|$ (< 100) and $|V|$ (10 to 100) are usually small. The actual complexity can be further reduced as we use $b = 2$ in Υ-value(G, b). In practice, it takes less than 100 seconds to generate londex$_m$ for the largest problems in the IPCs that we tested.

Table 4.2 shows the improvement of londex$_m$ over londex$_1$ regarding the average distance of the constraints in several representative problem instances. We see that for fact londex, londex$_m$ improves the average distance by 6% (Depot domain) to 36% (Zenotravel domain). The action londex is usually of much larger quantity, in tens of millions. We can also observe similar improvements in the action londex, mostly around 10%.

Algorithm 7: $generate_londex_m()$

Input: The set of all DTGs $\mathcal{G}$; The set of all facts $\mathcal{F}$

Output: all londex$_m$

```
1  foreach G = (V, E) do
2      foreach pair of facts (f, g) ∈ V^2 do
3          compute Δ_G(f, g) ;
4  foreach G = (V, E) do
5      call ϒ-value(G, b), where b=2;
6  foreach G = (V, E) do
7      foreach pair of facts (v, w) ∈ V^2 with Δ_G(v, w) ≥ 3 do
8          foreach bridge pair (f, g) of v ⇝ w do
9              if ϒ(f, g) > Δ_G(f, g) then
10                 increase the distance of (v,w) to Δ_G(v, f) + ϒ(f, g) + Δ_G(g, w) ;
11 generate londex_m for actions;
```

Problem	Fact londex			Action londex		
	Count	londex$_1$	londex$_m$	Count	londex$_1$	londex$_m$
Depot 20	27824	1.806	1.917	1799312	2.826	2.939
Driverlog 20	52242	2.401	2.792	2195384	2.119	2.301
Pipesworld 20	10672	1.887	2.188	14317438	2.023	2.257
Trucks 20	1628	1.926	2.176	28345668	4.027	4.447
Zenotravel 20	20070	1.612	2.187	30572950	2.593	2.723

Table 4.2: Comparisons of the average constraint distances for both fact londex and action londex. Column "Count" indicates the number of constraints we can derive in each problem. Columns 'londex$_1$' and 'londex$_m$' give the average constraint distances of londex$_1$ and londex$_m$, respectively.

4.3 Non-Clausal Londex Constraints

Londex constraints, in particular londex$_m$ constraints, has the disadvantage that it may substantially increase the encoding size. For instance, tens of millions of clauses may be generated from the londex$_m$ constraints. As a result, memory becomes a limiting factor for applying londex constraints.

Not all londex constraints are needed for constraint propagation during SAT solving. In fact, less than 1% of the londex constraints are used in the problem instances that we have experimented with. Thus, it is a waste of time and memory to generate and store those londex constraints that are never needed. However, it is difficult to determine or predict, at the encoding phase, which constraints are useful.

To address this memory issue, we propose a new framework of SAT-based planning in which we use londex constraints as nonclausal constraints. In the new approach, we do not encode any londex constraint as a SAT clause in the initial encoding phase, but rather instantiate those londex constraints that are needed on-the-fly during SAT solving in a conflict-driven way. The SAT solver used cannot be a blackbox but needs to be integrated with a londex reasoning mechanism. By using this approach, in most cases we only need to trigger less than 1% of all londex constraints, which are critically helpful. We can solve many planning instances in various domains that were not solvable previously due to the memory restriction.

4.3.1 Londex as Nonclausal Constraints

Our algorithm is specified in DPLL_nonclausal_londex() in Algorithm 5, which is a variant of the DPLL procedure used by MiniSat [34] integrated with londex as nonclausal constraints. An essential element of this algorithm is the method to identify and invoke required londex constraints to strengthen constraint propagation, particularly unit propagation, in SAT solving. If any conflict occurs while propagating (londex) constraints, we generate and add new nonclausal constraints to the SAT solver [34].

An action londex constraint can be represented in a *nonclausal form* $t(a) - t(b) \geq r$, where a and b are actions. To use the londex constraints in a SAT planner where the SAT solver is used as a blackbox, we need to convert a londex constraint in the nonclausal form into SAT clauses. We need to generate a clause $\neg v_{a,t} \vee \neg v_{a,t'}$, for any $t - t' < r$, $1 \leq t, t' \leq L$, where L is the total number of time steps.

In our new approach, we do not instantiate these clauses. Instead, we only save the londex constraints in their nonclausal forms. Let $\overline{r}$ be the average value of distance in the londex constraints, the space saving by using nonclausal forms is of the order $\Theta(\overline{r}L)$.

The original DPLL algorithm, used in Algorithm 5, works as follows. In each iteration, it selects an unassigned variable x and sets it to 1 (Line 6). MiniSat uses a heuristic [34] that orders the unassigned variables by their degrees of activities and chooses the one with the highest degree. After x is assigned to 1, it performs unit propagation in the standard DPLL algorithm (Line 14). During the unit propagation, an implied literal can be set to 1 or 0. The literals that are processed by the unit propagation enter the queue Q (Line 12). During the propagation, a conflict occurs if a propagated value or implied variable assignment is in conflict with a previous assignment. If a conflict is encountered, new clauses specifying the conflict, which has been often called "no-good" clause learning to avoid encountering the same conflict multiple times, are added to the SAT

Algorithm 8: DPLL_nonclausal_londex()

Input: variables $\mathbb{V}$, clauses $\mathbb{C}$, londex constraints $\mathbb{L}$

Output: return the solution to SAT problem or report no solution found

```
1  Q ← empty queue, conflict-free ← true;;
2  while true do
3      if conflict-free then
4          if no unassigned variable can be found then return satisfiable;
5          select the unassigned variable z with the highest activity;
6          enqueue(Q, ⟨z, value[z] = 1⟩);
7      else
8          if the conflict is found at the root level then return unsatisfiable;
9          analyze conflicts, generate learnt clauses and backtrack until conflicts are eliminated;
10     while Q is not empty do
11         ⟨x, value[x]⟩ = dequeue(Q);
12         propagate value[x], enqueue new assignments to Q, and break if conflict detected;
13         if value[x] = 1 then
14             for all literals y that must be 0 according to londex constraints do
15                 if y is already assigned as 1 then
16                     set conflict-free = false;
17                     break;
18                 else
19                     assign value[y] = 0;
20                     enqueue(Q, ⟨y, value[y] = 0⟩ ;
```

formulation and the algorithm backtracks to resolve the conflict (Line 9). Details of the backtracking process can be found in the paper describing MiniSat [34].

After the original unit propagation, if there is any conflict, we will perform an additional *londex-constraint propagation* if the newly dequeued or chosen variable x is assigned to 1 (Lines 15–20). When x has the value 1, the corresponding action or fact is placed at a certain time step. We check all of the londex constraints and identify all actions and facts that cannot occur at certain time steps. For example, if there is a londex constraint $t(a) - t(b) \geq 6$ for two actions a and b, and if $x = 1$ corresponds to placing a at time step 12, we set to 0 all those literals that correspond to placing b at time steps 7 to 12. These potential assignments in the form of $value[y] = 0$ will also be enqueued and further propagated (Lines 19–20).

We do not perform any londex-constraint propagation if x is set to 0 because $x = 0$ means a fact or action is not at a specific time step. Such a fact cannot be propagated using londex.

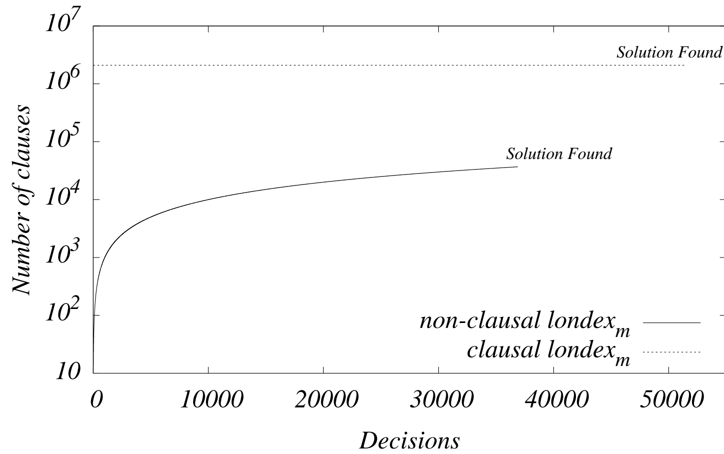


Figure 4.7: The numbers of londex clauses used by both clausal and nonclausal methods on Storage-15 (Makespan 8).

It is also possible that a conflict may be detected during the londex-constraint propagation. Since we always set an implied literal y to 0 during londex-constraint propagation, a conflict happens if y has already been assigned to 1. Such a conflict will be resolved in the same way as the original conflicts in the DPLL algorithm (Lines 15–17).

The algorithm terminates when there is no unassigned variable, in which case the problem is satisfiable (Line 4), or when a conflict is found at the root level during backtracking, in which case the problem is unsatisfiable (Line 8).

4.3.2 Effects of Nonclausal Londex Constraints

The new approach can reduce the memory usage by only enforcing on the fly a small portion of all londex constraints. For each constraint, it takes extra time to expand the londex constraints in nonclausal form before being used in constraint propagation. Surprisingly, we found that for many problems, this approach can save not only space but also search time. This is because the cost for checking and processing clauses in constraint propagation is greatly reduced since many fewer clauses are in the SAT encoding. In the experimental results section, we will show that the nonclausal londex approach can not only address the memory issue, but also make the algorithm faster and applicable to many large problem instances.

By using this approach, in most cases we only need to activate less than 1% of all londex constraints. Figure 4.7 compares the total number of londex constraint clauses used by the original method and the number of londex constraint clauses actually used by the new nonclausal approach on the

Storage-15 problem in IPC5. It is a representative example that the non-clausal method finds a solution with much fewer decisions than the original one. We can find similar cases in many other instances.

In the figure, we show the total number of londex constraint clauses used as the search algorithm proceeds. The label "solution found" marks the time when the SAT instance is solved. The original method instantiates all londex constraints as clauses in the SAT formulation and thus maintains a constant number of clauses. The new nonclausal method instantiate clauses on demand and exhibits a dramatic reduction on the number of clauses. It uses two orders of magnitude fewer clauses and solves the problem faster than the clausal londex approach. Using the new approach, we can solve many new planning instances in various domains, which were not solvable previously due to memory or time limitation.

4.4 Experimental Results

We now evaluate the effects of londex by integrating londex constraints in both SATPlan04 and SATPlan06. As discussed earlier, SATPlan04 and SATPlan06 differ mainly in their encoding mechanisms; SATPlan04 uses an action-based encoding with action literals only, while SATPlan06 uses a hybrid encoding that includes both action and fact literals. We show that londex constraint is effective in reducing the solution time for both encodings. We also show that londex$_m$ constraint is more powerful than londex$_1$ constraint. The experiments are conducted on a Xeon 2.0 GHz workstation with 2Gb of memory limit.

In our experiments, we study the performance of original SATPlan04 (denoted as SAT04), SATPlan04 with londex$_1$ as clausal constraints (denoted as A(1)), SATPlan04 with londex$_m$ as clausal constraints (denoted as A(m)), and SATPlan04 with londex$_m$ as nonclausal constraints (denoted as A(m)*). Here, "A" means using an action-based encoding.

We also integrate londex constraints into SATPlan06, which uses a hybrid encoding with action and fact literals. In our experiments, we compare the performance of the original SATPlan06 (denoted as SAT06), SATPlan06 with londex$_1$ as clausal constraints (denoted as H(1)), and SATPlan06 with londex$_m$ as clausal constraints (denoted as H(m)). Here, "H" means using a hybrid encoding.

In our experiments, we do not apply the nonclausal-constraint technique to the hybrid encoding for the following reasons. The motivation of the nonclausal constraints is to reduce memory consumption. For SATPlan04, since it uses an action-based encoding, we can only use action londex constraint, which is typically of very large quantity. For SATPlan06 which uses a hybrid encoding,

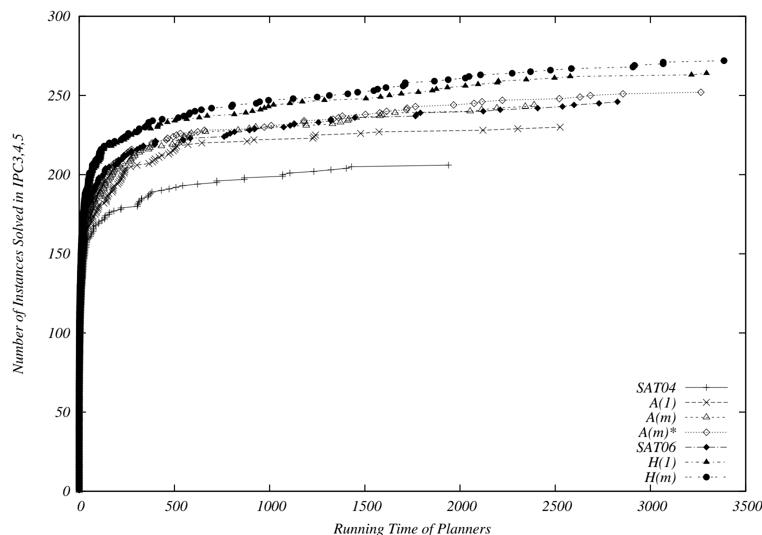


Figure 4.8: Number of instances solved by each planner.

since both fact and action literals are available, we can use either fact londex or action londex, or both. According to our experiments, adding both of fact and action londex constraints provides almost no extra benefit. The strong correlation between these two kinds of constraints make their pruning capability overlapped, as in general we can derive action londex from fact londex, and vice versa. On the other hand, since there are typically much fewer facts than actions in a planning instance, the quantity of fact londex constraints is much fewer than that of action londex constraint. Therefore, for the hybrid encoding in SATPlan06, memory is not a bottleneck and hence using londex as nonclausal constraints is not beneficial. Further, since the number of fact londex constraints is relatively very small, using fact londex as nonclausal constraints can save only negligible overhead for constraint propagation and we cannot observe any difference in the runtime. That is why we do not use nonclausal constraints for the hybrid encoding.

We conduct the experiments on all of the IPC3, IPC4 and IPC5 domains. In our experiments, if a domain appears in more than one IPC, we use the one in the latest IPC. Figure 4.8 illustrates the number of instances solved by each planner when the solving time increases. We see from Figure 4.8 that the efficiency of the seven planners can be ranked as, from best to worst, H(m), H(1), A(m)*, A(m), SAT06, A(1) and SAT04. For each problem instance, we have verified that all solvers give the same makespan.

On the IPC domains that we test, SAT04 has the worst performance. A(1) is much better than SAT04, where londex constraints greatly helps the performance. The encoding of SAT06 is better overall. In addition, while londex constraints are integrated, we achieve moderate improvements.

4.5 Summary

Londex is a general class of constraints that can be automatically derived from the problem structure of STRIPS planning domains. We have first proposed londex_1, which is derived based on the topology of individual DTGs. Londex_1 gives rise to state-independent minimum distance of actions and facts that can be utilized during the planing process. We further extend londex_1 by exploiting the causal dependencies among multiple DTGs. The resulting londex_m provides tighter lower bounds on the minimum state-independent distances between facts and actions, leading to stronger search space pruning.

We have integrated londex into SAT-based planning. By incorporating londex constraints into the SAT formulation, we are able to achieve strong constraint propagation and significant improvement to planning efficiency. In order to ease the burden of a high memory requirement by londex, we have proposed a mechanism for utilizing the londex constraints as nonclausal constraints. Instead of adding londex constraints as clauses to the SAT encoding, we have modified the DPLL search algorithm and used londex for unit propagation in a conflict-driven fashion so as to generate only the londex constraints as needed. This technique enables us to make full use of the pruning power of londex without exhausting available memory.

The experimental results on recent IPC domains show that londex constraints can speed up planners using both action-based and hybrid SAT encodings, on most problems of nearly all domains that we tested.

Although londex has achieved extensive improvement regarding problem solving efficiency, there are a few limitations. First, we have to keep two sets of planning formulations in a planner: STRIPS for encoding and SAS+ for londex constraints. This leads to additional processing time. Second, this method is in general not memory efficient. As we will show in the next chapter, the STRIPS based encoding is in fact not compact enough. Thus having more redundant clauses makes it even more memory consuming. Having too many clauses not only have side effects on the memory usage, but also makes the SAT solving less efficient in some instances.

Chapter 5

SAS+ Planning as Satisfiability

A key factor for the performance of the planning as satisfiability approach is the SAT encoding scheme, which is the way a planning problem is compiled into SAT formulae with boolean variables and clauses. As the encoding scheme has a great impact to the efficiency of SAT-based planning, developing novel and superior SAT encoding has been an active research topic. Previously, extensive research has been done on making the SAT encoding more compact. One example of compact encoding is the lifted action representation, first studied in [72] and soon later more comprehensively in [35]. In this compact encoding scheme, actions are represented by a conjunction of parameters, thus this method mitigates the problem of blowing up time steps caused by grounding and itemizing each action. The original scheme does not guarantee the optimality on time steps, however, an improved lifted action representation that preserves optimality is proposed[109]. Later, a new encoding [107] is also proposed based on a relaxed parallelism semantic, which also does not guarantee optimality.

All these previous enhancements were still based on STRIPS. In this chapter, we propose the first SAS+ based encoding scheme (SASE) for classical planning. Unlike previous STRIPS based SAT encoding schemes that model actions and facts, SASE directly models *transitions* in the SAS+ formulation. Transitions can be viewed as a high-level abstraction of actions, and there are typically significantly fewer transitions than actions in a planning task. The proposed SASE scheme describes two major classes of constraints: first the constraints between transitions and second the constraints that match actions with transitions. We theoretically and empirically study SASE and compare it against the transitional STRIPS based SAT encoding. To further improve the performance of SASE, we propose a number of techniques to reduce encoding size by recognizing certain structures of actions and transitions in it.

Theoretically, we study the relationship between the solution spaces of SASE and that of STRIPS based encoding. We show that the set of solution plans found by the STRIPS based encoding and

by SASE are isomorphic, meaning that there is a bijective mapping between the two. Hence, we show the equivalence between solving the STRIPS based encoding and SASE.

As an attempt to assess the performance gain of SASE, we study how it makes a DPLL SAT solving algorithm behave in a more favorable way. The study is quantified by the widely used VSIDS heuristic. The transition variables that we introduce have high frequencies in clauses. Thus transition variables consequently have higher VSIDS scores. The higher VSIDS scores lead to more branching on the transition variables than action variables. Since the transition variables has high scores and hence stronger constraint propagation, branching more on the transition variables leads to faster SAT solving. We provide empirical evidences to support our explanation. More importantly, we introduce an indicator called transition index, and empirically show that there is strong correlation between the transition index and SAT solving speedup.

Finally, we evaluate the new encoding on the standard benchmarks from the recent International Planning Competitions. Our results show that the new SASE encoding scheme is more efficient in terms of both time and memory usage comparing to STRIPS-based encodings, and solves some large instances that the state-of-the-art STRIPS-based SAT planners fail to solve.

We first present SASE in Section 5.1 and prove its equivalence to the STRIPS based encoding in Section 5.2. We study the reason why SASE works better in modern SAT solving algorithms, by conducting empirical studies to support the worst case analysis in Section 5.3. The techniques to further reduce the encoding size are considered in Section 5.4. We present our experimental results in Section 5.5, and summarize in the last section.

5.1 SASE Encoding Scheme

In this section, we introduce our new encoding for SAS+ planning tasks, denoted as SASE. We use the same search framework as SatPlan: start with a small number of time steps N and increase N by one each time until a satisfiable solution is found. For each given N, we encode a planning task into a SAT instance which can be solved by a SAT solver. A SASE instance includes two types of binary variables:

1. Transition variables: $U_{\delta,t}$, $\forall \delta \in \mathcal{T}$ and $t \in [1, N]$, which may also be written as $U_{x,f,g,t}$ when δ is explicitly $\delta^x_{f \to g}$;
2. Action variables: $U_{a,t}$, $\forall a \in \mathcal{O}$ and $t \in [1, N]$.

As to constraints, SASE has eight classes of clauses for a SAS+ planning task. In the following, we define each class for every time step $t \in [1, N]$ unless otherwise indicated.

1. Initial state: $\forall x, s_{\mathcal{I}}(x) = f$, $\bigvee_{\forall \delta_{f \to g} \in \mathcal{T}(x)} U_{x,f,g,1}$;
2. Goal: $\forall x, s_{\mathcal{G}}(x) = g$, $\bigvee_{\forall \delta_{f \to g} \in \mathcal{T}(x)} U_{x,f,g,N}$;
3. Progression: $\forall \delta^x_{h \to f} \in \mathcal{T}$ and $t \in [1, N-1]$, $U_{x,h,f,t} \to \bigvee_{\forall \delta^x_{f \to g} \in \mathcal{T}(x)} U_{x,f,g,t+1}$;
4. Regression: $\forall \delta^x_{f \to g} \in \mathcal{T}$ and $t \in [2, N]$, $U_{x,f,g,t} \to \bigvee_{\forall \delta^x_{f' \to f} \in \mathcal{T}(x)} U_{x,f',f,t-1}$;
5. Transition mutex: $\forall \delta_1 \forall \delta_2$ such that δ_1 and δ_2 are transition mutex, $\overline{U}_{\delta_1,t} \vee \overline{U}_{\delta_2,t}$;
6. Composition of actions: $\forall a \in \mathcal{O}$, $U_{a,t} \to \bigwedge_{\forall \delta \in M(a)} U_{\delta,t}$;
7. Action existence: $\forall \delta \in \mathcal{T} \setminus R$, $U_{\delta,t} \to \bigvee_{\forall a, \delta \in M(a)} U_{a,t}$;
8. Action mutex: $\forall a_1 \forall a_2$ such that $\exists \delta, \delta \in T(a_1) \cap T(a_2)$ and $\delta \notin R$, $\overline{U}_{a_1,t} \vee \overline{U}_{a_2,t}$;

Clauses in classes 3 and 4 specify and restrict how transitions change over time. Clauses in classes 5 enforce that at most one related transition can be true for each state variable at each time step. Clauses in classes 6 and 7 together encode how actions are composed to match the transitions. Clauses in class 8 enforce mutual exclusion between actions.

Note there are essential differences between transition variables in SASE and fact variables in PE. In terms of semantics, a transition variable at time step n in SASE is equivalent to the conjunction of two fact variables in PE, at time step n and $n+1$, respectively. Nevertheless, facts variables are not able to enforce a transition plan as transition variables do. The reason is that transition variables not only imply the values of multi-valued variables, but also enforce how these values propagate over time steps.

In addition, transition variables are different from action variables regarding their roles in SAT solving. The reason is as follows. In SASE, action variables only exist in the constraints for transition-action matching, but not in the constraints between time steps. Transition variables exist in both. Thus transition variables appear more frequently in a SAT instance. The inclusion of those high-frequency variables can help the SAT solvers through the VSIDS rule for variable branching. We will discuss more and provide an empirical study in Section 5.3.

5.1.1 Search Spaces of Encodings

It is in general difficult to accurately estimate the time that a SAT solver needs to solve a SAT instance, as it depends on not only the problem size, but also the structure of the clauses. In this section, we give a preliminary analysis of the worst case search space of a planning problem encoded in SASE for a given time step N. In particular, we examine the search spaces corresponding to the SAT instances in PE and SASE, respectively. We analyze the underlying encoding structures in SASE and why a problem in SASE can be typically efficiently solved.

We first consider the search space of planning in PE. To simplify the analysis, we focus on an action-based encoding. The argument can be readily extended to an encoding with both actions and facts explicitly represented. In an action-based encoding, one binary variable is introduced for each action a at a time step t. The constraint propagation is achieved through application of actions in this encoding; hence, a key problem is to select a set of actions for each time step t. There are $2^{|\mathcal{O}|}$ possible subsets of actions at each time step. Therefore, a total of $(2^{|\mathcal{O}|})^N$ possible action plans in the search space. An exhaustive search will explore a search space of size $O((2^{|\mathcal{O}|})^N)$.

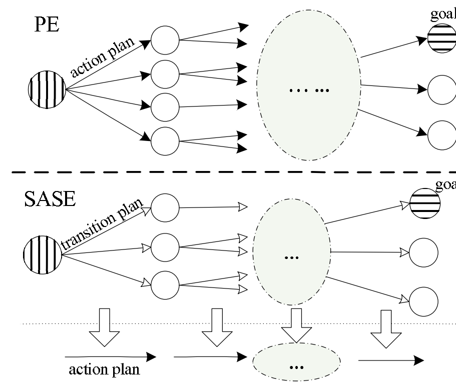


Figure 5.1: Illustration of how the search spaces of two encoding schemes differ from each other.

The major difference between the SAT instances in SASE and PE is that in the former encoding, actions are not responsible for the constraint propagation across time steps. Figure 5.1 illustrates their difference. In SASE, the SAT instance can be conceptually reduced to the following search problem of two hierarchies.

- At the top level, we search for a transition plan as defined in Definition 4. This amounts to finding a set of transitions for each time step t (corresponding to all δ such that $U_{\delta,t}$ is set to $\top$), so that they satisfy the clauses in classes 1-5 of the SASE encoding.

- At the lower level, we try to find an action plan that satisfies the transition plan. In other words, for a given transition plan that satisfies clauses in classes 1-5, we try to find an action plan satisfying clauses in classes 6-8.

We now analyze the search space size in both hierarchies of SASE. For the top level, since there are $|\mathcal{T}|$ transitions, at each time step we has $2^{|\mathcal{T}|}$ choices thus the size is in total $(2^{|\mathcal{T}|})^N$. We note that $|\mathcal{T}|$ is usually much less than $|\mathcal{O}|$. On the lower level, two observations can be made. For a time step t and a given subset of selected transitions (corresponding to all δ such that $U_{\delta,t}$ is set to 1), finding a subset of actions that satisfies clauses in classes 6-8 amounts to exploring a search space with size $K = \prod_{\delta \in \mathcal{T}} |A(\delta)|$ in the worst case. Given a transition plan, the problems of finding a supporting action plan at different time steps are independent of one another. That is, an action plan can be found for each time step separately without backtracking across different time steps. Hence, the total cost of the lower level is NK. Therefore, to solve an SASE instance, the search space that an exhaustive search may explore is bounded by $O((2^{|\mathcal{T}|})^N NK)$.

The number of transitions $|\mathcal{T}|$ is generally much smaller than the number of actions $|\mathcal{O}|$ in practise. For instance, in Pipesworld-30, $|\mathcal{O}|$ is 15912 and $|\mathcal{T}|$ is 3474; in TPP-30, $|\mathcal{O}|$ is 11202 and $|\mathcal{T}|$ is 1988. On the other hand, although K is exponential in $|\mathcal{T}|$, it is a relatively smaller term. Therefore, the bound of SASE $O((2^{|\mathcal{T}|})^N NK)$ is smaller than the one for STRIPS-based encoding $O((2^{|\mathcal{O}|})^N)$.

5.1.2 A Running Example

Let us show how SASE works on an example. Consider a planning task with two multi-valued variables x and y, where $dom(x) = \{f, g, h\}$ and $dom(y) = \{d, e\}$. There are three actions $a1 = \{\delta^x_{f\to g}, \delta^y_{d\to e}\}$, $a2 = \{\delta^x_{f\to g}, \delta^y_{e\to d}\}$ and $a3 = \{\delta^x_{g\to h}, \delta^y_{e\to d}\}$. The initial state is $\{x = f, y = d\}$ and the goal state is $\{x = h, y = d\}$. One solution to this instance is a plan of two actions: $a1$ at time step 1 and then $a3$ at time step 2.

In the following we list the constraints between transitions and actions, namely those specified in classes 6 and 7. The clauses in other classes are self-explanatory. In particular, here we only list the variables and clauses for time step 1, because these constraints all repeat for time step 2. The transition variables at time step 1 are $\{$ $U_{x,f,g,1}$, $U_{x,f,f,1}$, $U_{x,g,h,1}$, $U_{x,g,g,1}$, $U_{x,h,h,1}$, $U_{y,d,d,1}$, $U_{y,e,e,1}$, $U_{y,e,d,1}$ $\}$, and they repeat for time step 2. The action variables at time step 1 are $\{$ $U_{a1,1}$, $U_{a2,1}$, $U_{a3,1}$$\}$, and they repeat for time step 2.

The clauses in class 6 are: $\overline{U}_{a1,1} \vee U_{x,f,g,1}$, $\overline{U}_{a1,1} \vee U_{x,d,e,1}$, $\overline{U}_{a2,1} \vee U_{x,f,g,1}$, $\overline{U}_{a2,1} \vee U_{y,e,d,1}$, $\overline{U}_{a3,1} \vee U_{x,g,h,1}$ and $\overline{U}_{a3,1} \vee U_{x,e,d,1}$. The clauses in class 7 are $\overline{U}_{x,f,g,1} \vee U_{a1,1} \vee U_{a2,1}$, $\overline{U}_{x,g,h,1} \vee U_{a3,1}$, $\overline{U}_{x,d,e,1} \vee U_{a1,1}$, and $\overline{U}_{x,e,d,1} \vee U_{a2,1} \vee U_{a3,1}$.

The solution, in terms of actions, has action variables $U_{a1,1}$ and $U_{a3,2}$ to be true, and all other action variables to be false. In addition, the corresponding transition plan has the following transition variables to be true: $\{U_{x,f,g,1}, U_{x,g,h,2}, U_{y,d,e,1}, U_{y,e,d,2}\}$, while all other transition variables are false.

As mentioned above, although there are often multiple transition plans, a transition plan may not correspond to a valid action plan. In this particular example, there are several different transition plans that satisfy the initial state and the goal, but some of them do not have a corresponding action plan. For example by having transition variables $\{U_{x,f,g,1}, U_{x,g,h,2}, U_{y,d,d,1}, U_{y,d,d,2}\}$ to be true, it results in a transition plan, but does not lead to a valid action plan.

5.2 Correctness of SAS+ Based Encoding

It is important to prove the correctness of the proposed encoding. We achieve so by proving that SASE for SAS+ planning has the same solution space as that of PE used in STRIPS planning. More specifically, we show that, for a given planning task and a given time step N, the SAT instance from SASE is satisfiable if and only if the SAT instance from PE is satisfiable. Here, we assume the correctness of the PE encoding, used by a number of planners such as SatPlan06 [74] for STRIPS planning.

Given a STRIPS task Ψ, we use PE(Ψ, N) to denote the encoding by PE with a time step bound N. Similarly, we use SASE(Π, N) to denote that by SASE, when given a SAS+ task Π.

5.2.1 Solution Structures of STRIPS Based Encoding

In this section, we study a few properties of the solutions in a STRIPS based encoding. These properties provide some key insights for establishing the relationship between PE and SASE encodings.

Lemma 1 *Given a STRIPS task $\Psi = (\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$, a time step N, and its* PE *SAT instance* PE$(\Psi, N) = (V, C)$*, suppose there are a satisfiable solution denoted as Γ, a fact $f \in \mathcal{F}$, and $t \in [1, N]$ such that: 1) $\Gamma(W_{dum_f,t}) = \perp$, 2) $\Gamma(W_{f,t}) = \top$, and 3) $\forall a \in$* DEL(f)*, $\Gamma(W_{a,t}) = \perp$,*

then we can construct an alternative solution Γ' *to* PE(Ψ, N) *as follows:*

$$\Gamma'(v) = \begin{cases} \top, v = W_{dum_f,t}, v \in V \\ \Gamma(v), v \neq W_{dum_f,t}, v \in V \end{cases} \quad (5.1)$$

Proof We show that Γ' satisfies every clause in C just as Γ does. Since all variables but $W_{dum_f,t}$ keep the same value, we only need to look at those clauses that have $W_{dum_f,t}$ in them. According to the definition of PE, $W_{dum_f,t}$ may exist in three types of clauses:

1. Clauses for add effects. In this case, the clauses are of the form $W_{f,t+1} \rightarrow (W_{dum_f,t} \vee W_{a_1,t} \vee \ldots \vee W_{a_m,t})$, which is equivalent to $\overline{W}_{f,t+1} \vee W_{dum_f,t} \vee W_{a_1,t} \vee \ldots \vee W_{a_m,t}$. Since $\Gamma'(W_{dum_f,t}) = \top$, such clauses are still true.
2. Clauses for preconditions. In this case, the clauses are of the form $W_{dum_f,t} \rightarrow W_{f,t}$, which is equivalent to $\overline{W_{dum_f,t}} \vee W_{f,t}$. Since $\Gamma'(W_{f,t}) = \top$, these clauses remain true for Γ'.
3. Clauses of mutual exclusion between actions. Without loss of generality, let us denote such a clause $\overline{W_{dum_f,t}} \vee \overline{W_{a,t}}$. For a given f, the actions in all such clauses are mutex with dum_f, because f is their delete effect. According to the construction, since $\Gamma'(W_{a,t}) = \Gamma(W_{a,t}) = \perp$, all such clauses are true.

The three cases above conclude that all clauses that include $W_{dum_f,t}$ are satisfied by Γ'. Therefore, Γ' is also a solution to PE. □

Lemma 2 *Given a STRIPS task* $\Psi = (\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$*, a time step* N*, and its* PE *SAT instance* PE$(\Psi, N) = (V, C)$*, suppose there are a satisfiable solution denoted as* Γ*, a fact* $f \in \mathcal{F}$*, and* $t \in [1, N]$ *such that: 1)* $\Gamma(W_{f,t}) = \perp$*, 2) there exists an action* $a \in$ ADD(f) *such that* $\Gamma(W_{a,t-1}) = \top$*, then we can construct an alternative solution* Γ' *to* PE(Ψ, N) *as follows:*

$$\Gamma'(v) = \begin{cases} \top, v = W_{f,t}, v \in V \\ \Gamma(v), v \neq W_{f,t}, v \in V \end{cases} \quad (5.2)$$

Proof We will show that Γ' makes each clause in C to be true. Since all variables but $W_{f,t}$ keep the same value, we only need to look at those clauses that have $W_{f,t}$ in them. According to the definition of PE, $W_{f,t}$ may exist in three types of clauses.

1. Clauses for add effects. In this case, f is an add effect of multiple actions. Let us write this clauses as $W_{f,t} \rightarrow (W_{a_1,t-1} \vee W_{a_2,t-1} \vee \ldots \vee W_{a_m,t-1})$, which is $\overline{W_{f,t}} \vee W_{a_1,t-1} \vee W_{a_2,t-1} \vee$

$\ldots \vee W_{a_m,t-1}$. Since there exists an action $a \in \text{ADD}(f)$ such that $\Gamma(W_{a,t-1}) = \top$, the clause is still true in Γ'.

2. Clauses for preconditions. In this case, f is a precondition of an action b. This clause is written as $W_{b,t} \to W_{f,t}$, which is equivalent to $\overline{W_{b,t}} \vee W_{f,t}$. Since $\Gamma'(W_{f,t}) = \top$, this clause is still true.

3. Clauses of fact mutex. Without loss of generality, consider a fact g that is mutex with f. The corresponding clause will be $\overline{W_{f,t}} \vee \overline{W_{g,t}}$. Since $\Gamma'(W_{f,t}) = \top$, this clause is true if $\Gamma'(W_{g,t}) = \perp$.

   We now suppose $\Gamma'(W_{g,t}) = \top$ and show that it leads to a contradiction. According to clauses of class III, there must be a variable $W_{b,t-1}$, such that $g \in add(b)$ and $\Gamma'(W_{b,t-1}) = \top$. According to the definition of mutex, two facts are mutex only when every pair of the actions that add them are mutex. Thus, $W_{a,t-1}$ and $W_{b,t-1}$ are mutex. Therefore, $\Gamma'(W_{a,t-1}) = \top$ and $\Gamma'(W_{b,t-1}) = \top$, leading to a contradiction. As a result, $\Gamma'(W_{g,t}) = \perp$, and consequently this clause is satisfied.

The three cases above conclude that all clauses that include $W_{f,t}$ are satisfied by Γ'. Therefore, Γ' is also a solution to PE. □

Lemmas 1 and 2 show that under certain conditions, some dummy action variables and fact variables in PE are *free variables*. They can be set to be either true or false while the SAT instance remains satisfied. Note that although we can manipulate these free variables to construct an alternative solution Γ' from a given solution Γ, both Γ and Γ' refer to the same STRIPS plan, because there is no change to any real action variable. This leads to an important insight on the solutions of PE: a solution plan to a STRIPS planning problem Ψ may correspond to multiple solutions to PE(Ψ, N).

Lemma 3 *Given a STRIPS task* $\Psi = (\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$*, a time step* N*, and its* PE *SAT instance* PE$(\Psi, N) = (V, C)$*, those clauses that define competing needs mutex and fact mutex can be inferred from other clauses in* PE(Ψ, N).

Proof We need to prove:

- if actions a and b are mutex at level t due to competing needs, the clause $\overline{W}_{a,t} \vee \overline{W}_{b,t}$ can be inferred from other clauses (except for the fact mutex clauses) in PE.
- if facts f_1 and f_2 are mutex at level t, the clause $\overline{W}_{f_1,t} \vee \overline{W}_{f_2,t}$ can be inferred from other clauses (except for the competing need mutex clauses) in PE.

We prove by mathematical induction on t. When $t = 0$, each two facts in the initial state are not mutex, thus there is no competing needs mutex and no fact mutex. Now suppose the hypothesis holds at time step t, we prove that the mutex clauses defined by competing needs mutex and fact mutex at time $t+1$ can be inferred from other clauses.

Consider two facts f_1 and f_2 that are mutex at level $t+1$. According to the definition of fact mutex, for every two actions a and b such that $a \in \text{ADD}(f_1), b \in \text{ADD}(f_2)$, a and b are mutex. Based on the induction, we have $\overline{W}_{a,t} \vee \overline{W}_{b,t}$ (for $\forall a, b$ that $a \in \text{ADD}(f_1)$, and $b \in \text{ADD}(f_2)$). Since $f_1 \in add(a)$ and $f_2 \in add(b)$, we have $W_{f_1,t+1} \rightarrow \bigvee_{\forall a \in \text{ADD}(f_1)} W_{a,t}$ and $W_{f_2,t+1} \rightarrow \bigvee_{\forall b \in \text{ADD}(f_2)} W_{b,t}$. A resolution using these three types of clauses can be performed as follows:

$$\frac{\overline{W}_{f_1,t+1} \vee \bigvee_{\forall a \in \text{ADD}(f_1)} W_{a,t},\ \overline{W}_{a,t} \vee \overline{W}_{b,t}\ (\forall a \in \text{ADD}(f_1), \forall b \in \text{ADD}(f_2))}{\overline{W}_{f_1,t+1} \vee \bigvee_{\forall b \in \text{ADD}(f_2)} \overline{W}_{b,t}} \tag{5.3}$$

Another resolution using (5.3) gives:

$$\frac{\overline{W}_{f_1,t+1} \vee \bigvee_{\forall b \in \text{ADD}(f_2)} \overline{W}_{b,t},\ \overline{W}_{f_2,t+1} \vee \bigvee_{\forall b \in \text{ADD}(f_2)} W_{b,t}}{\overline{W}_{f_1,t+1} \vee \overline{W}_{f_2,t+1}} \tag{5.4}$$

Hence, fact mutex at level $t+1$ can be inferred. Now, consider two actions a and b at level $t+1$ which are mutex because of competing needs. There exist two mutex facts $f \in pre(a)$ and $g \in pre(b)$. From above, we have $\overline{W}_{f,t+1} \vee \overline{W}_{g,t+1}$. Since $f \in pre(a)$ and $g \in pre(b)$, then we have $W_{a,t+1} \rightarrow W_{f,t+1}$ and $W_{b,t+1} \rightarrow W_{g,t+1}$.

We can perform a resolution using these three types of clauses:

$$\frac{\overline{W}_{a,t+1} \vee W_{f,t+1},\ \overline{W}_{f,t+1} \vee \overline{W}_{g,t+1}}{\overline{W}_{a,t+1} \vee \overline{W}_{g,t+1}} \tag{5.5}$$

Another resolution using (5.5) gives:

$$\frac{\overline{W}_{b,t+1} \vee W_{g,t+1},\ \overline{W}_{a,t+1} \vee \overline{W}_{g,t+1}}{\overline{W}_{a,t+1} \vee \overline{W}_{b,t+1}}. \tag{5.6}$$

Therefore, the action mutex defined by *competing needs* can also be inferred from other clauses. □

Lemma 3 shows that when encoding a STRIPS task, it is not necessary to encode fact mutex and competing needs action mutex, as they are implied by other clauses. Therefore, when we consider the completeness and correctness of PE, we can ignore these redundant clauses. Analysis with similar conclusion can be found in some literatures [116], but using different approaches.

5.2.2 Equivalence of STRIPS and SAS+ Based Encodings

A classical planning problem can be represented by both STRIPS and SAS+ formalisms, in which case we say the two formalisms are **equivalent**. Given a STRIPS task $\Psi = (\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$ and its equivalent SAS+ planning task $\Pi = (\mathcal{X}, \mathcal{O}, s_{\mathcal{I}}, s_{\mathcal{G}})$, the following isomorphisms (bijective mappings) exist:

- $\phi_f : \mathcal{F} \to \prod_{\mathcal{X}} Dom(X)$ (a binary STRIPS fact corresponds to an variable assignment in SAS+);
- $\phi_a : \mathcal{A} \to \mathcal{O}$ (a STRIPS action corresponds to a SAS+ action);
- $\phi_i : \varphi_{\mathcal{I}} \to s_{\mathcal{I}}$ (can be derived from ϕ_f);
- $\phi_g : \varphi_{\mathcal{G}} \to s_{\mathcal{G}}$ (can be derived from ϕ_f).

Furthermore, since both formalisms represent the same planning task, these mappings preserve the relations between actions and facts. For example, if $f \in pre(a)$ where $f \in \mathcal{F}$ and $a \in \mathcal{A}$ in a STRIPS formalism, we have $\phi_f(f) \in pre(\phi_a(a))$ in the SAS+ formalism.

First, we show that the parallelism semantics enforced by S-mutex in SAS+ is equivalent to that by P-mutex in STRIPS.

Lemma 4 *Given a SAS+ planning task* $\Pi = (\mathcal{X}, \mathcal{O}, s_{\mathcal{I}}, s_{\mathcal{G}})$ *and its equivalent STRIPS task* $\Psi = (\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$*, suppose we have actions* $a, b \in \mathcal{O}$*, and their equivalent actions* $a', b' \in \mathcal{A}$ *(i.e.* $a = \phi_a(a')$ *and* $b = \phi_a(b')$*),* a *and* b *are S-mutex if and only if* a' *and* b' *are P-mutex.*

Proof We construct the proof by studying it on both directions. Based on Lemma 3, we only consider *inconsistent effects* and *interference* mutex in P-mutex.

$\Rightarrow$*: if* a' *and* b' *are P-mutex in* Ψ*,* a *and* b *are S-mutex in* Π.

Since a' and b' are P-mutex, one either deletes precondition or add-effect of the other. Without loss of generality, suppose a' deletes f (i.e. $f \in del(a') \cap pre(b')$). Consequently, there must be a

transition $\delta_1 = \delta^x_{f\to h} \in T(a)$ such that $f \neq h$ and $\delta_2 = \delta^x_{f\to g} \in T(b)$. There are two cases to be considered.

1) $\delta_1 \neq \delta_2$. δ_1 and δ_2 are mutex transitions by Definition 3, since they both transit from f. Therefore, a and b are S-mutex, according to the second condition in Definition 5.

2) $\delta_1 = \delta_2$. In this case, a and b are S-mutex by the first condition of Definition 5.

Based on the two cases, we conclude that a and b are S-mutex. A similar argument applies to the case when one action deletes the other's add-effect.

$\Leftarrow$: *if a and b are S-mutex in Π, a' and b' are P-mutex in Ψ.*

If two actions a and b are S-mutex in Π, there are two cases.

1) There exists a transition δ, which is in both $T(a)$ and $T(b)$. Consequently, a' and b' deletes each other's precondition and thus they are P-mutex.

2) There exist two distinct transitions $\delta_1 \in T(a), \delta_2 \in T(b)$ and a multi-valued variable $x \in \mathcal{X}$, such that $\{\delta_1, \delta_2\} \subseteq T(x)$. Let us denote these two transitions as $\delta^x_{v_1\to v_2}$ and $\delta^x_{v_3\to v_4}$. In such a case, suppose $\delta^x_{v_1\to v_2}$ and $\delta^x_{v_3\to v_4}$ are allowed to be executed in parallel in a STRIPS plan. It obviously leads to a contradiction, since $v_1, v_2, v_3, v_4 \in Dom(x)$ are values of the same multi-valued variable, and by the definition of SAS+ formalism, only one of them can be true at the same time. Therefore, the preconditions of a' and b' must be mutex, and hence a' and b' are P-mutex. □

Lemma 4 gives the connection between P-mutex and S-mutex. Based on that we can construct the relations between the encodings, which are presented in Theorem 1 and Theorem 2, respectively.

Theorem 1 *Given a STRIPS task Ψ and a SAS+ task Π that are equivalent, for a time step bound N, if* PE(Ψ, N) *is satisfiable,* SASE(Π, N) *is also satisfiable.*

Proof Since PE(Ψ, N) is satisfiable, we denote one of its solutions as Γ_Ψ. We first present how to construct an assignment to SASE(Π, N) from Γ_Ψ. Next, we prove that this constructed assignment satisfies every clause in SASE(Π, N).

Construction. There are two steps for the construction. According to Lemmas 1 and 2, there are in general some free variables in Γ_Ψ. In the first step, we construct an alternative solution to PE(Ψ, N) by changing all free variables in Γ_Ψ to be true according to Lemmas 1 and 2. Let us denote the resulting solution as Γ'_Ψ. Then, we construct an assignment for SASE(Π, N) from Γ'_Ψ. The value of each variable in Γ_Π is defined as follows.

1. For every $a \in \mathcal{O}$ (which is also in $\mathcal{A}$)[1], we let $U_{a,t} = W_{a,t}$.
2. For every transition $\delta_{f \to g} \in \mathcal{T}$, if $W_{f,t} = \top$ and $W_{g,t+1} = \top$ in Γ'_Ψ, we set $U_{x,f,g,t} = \top$ in Γ_Π.

Satisfiability. We prove that every individual clause in SASE is satisfied by Γ_Π. There are eight types of clauses.

1. (Forward progression). According to our construction, we need to show that, for any $t \in [1, N-2]$,

$$\forall \delta^x_{h \to f} \in \mathcal{T}, (W_{h,t} \wedge W_{f,t+1}) \to \bigvee_{\forall g, \delta^x_{f \to g} \in \mathcal{T}} (W_{f,t+1} \wedge W_{g,t+2}) \tag{5.7}$$

If $\Gamma'_\Psi(W_{f,t+1}) = \perp$, then (5.7) is satisfied by Γ'_Ψ. If $\Gamma'_\Psi(W_{f,t+1}) = \top$, we consider an action set $Y = \{dum_f\} \cup \text{DEL}(f)$, which is a subset of $M(\delta^x_{f \to g})$. There are two possibilities.

- For every action $a \in Y$, $\Gamma'_\Psi(W_{a,t+1}) = \perp$. In such a case, $W_{dum_f,t+1}$ and $W_{f,t+2}$ are free variables according to Lemmas 1 and 2, respectively. Therefore, according to the construction in Γ'_Ψ, which assigns all free variables to true, variables $W_{f,t+1}$, $W_{f,t+2}$ and $W_{dum_f,t+1}$ are all $\top$. In addition, $\delta_{f \to f}$ is always in $\mathcal{T}$, meaning $W_{f,t+2}$ is included in the right hand side of (5.7). Therefore, (5.7) is satisfied by Γ'_Ψ.
- There exists an action $a \in Y$, such that $\Gamma'_\Psi(W_{a,t+1}) = \top$. In such a case, let us consider an arbitrary fact $g \in add(a)$. If $\Gamma'_\Psi(W_{g,t+2}) = \top$, then (5.7) is satisfied by Γ'_Ψ. Otherwise, according to Lemma 2, $W_{g,t+2}$ is a free variable and $W_{g,t+2}$ is already set to true in our construction of Γ'_Ψ. Therefore, Γ'_Ψ satisfies (5.7).

2. (Regression). According to our construction, we need to show that, for any $t \in [2, N-1]$,

$$\forall \delta^x_{f \to g} \in \mathcal{T}, (W_{f,t} \wedge W_{g,t+1}) \to \bigvee_{\forall h, \delta^x_{h \to f} \in \mathcal{T}(x)} (W_{h,t-1} \wedge W_{f,t}) \tag{5.8}$$

Consider clauses of class III (add effect) in PE. These clauses indicate that for each fact $f \in \mathcal{F}$, $W_{f,t}$ implies a disjunction of $W_{a,t-1}$ for all actions a such that $f \in add(a)$. Thus, for a given f, the following clauses are included in PE, which are satisfied by Γ'_Ψ:

$$W_{f,t} \to \bigvee_{\forall a \in \text{ADD}(f)} W_{a,t-1}. \tag{5.9}$$

[1]For simplicity, we use a to denote the same action in $\mathcal{A}$ instead of using $\phi_a(a)$.

For a given f, we consider the action set $\bigcup_{\forall h} A(\delta_{h\to f})$, denoted as Z. Since $\text{ADD}(f) \subseteq Z$,

$$W_{f,t} \to \bigvee_{\forall a \in Z} W_{a,t-1} \tag{5.10}$$

For any transition $\delta^x_{h\to f}$, for each action $a \in A(\delta^x_{h\to f})$, since $h \in pre(a)$, Γ'_Ψ satisfies $W_{a,t-1} \to W_{h,t-1}$. Therefore, for each $h \in pre(a)$, we have

$$\bigvee_{\forall a \in A(\delta^x_{h\to f})} W_{a,t-1} \to W_{h,t-1}. \tag{5.11}$$

Expand the set Z, we can convert (5.10) to:

$$W_{f,t} \to \bigvee_{\forall h, \delta^x_{h\to f} \in \mathcal{T}} \Big(\bigvee_{\forall a \in A(\delta^x_{h\to f})} W_{a,t-1} \Big). \tag{5.12}$$

By combining (5.11) and (5.12), we have:

$$W_{f,t} \to \bigvee_{\forall h, \delta^x_{h\to f} \in \mathcal{T}} W_{h,t-1}, \tag{5.13}$$

which implies

$$W_{f,t} \to \bigvee_{\forall h, \delta^x_{h\to f} \in \mathcal{T}} (W_{h,t-1} \wedge W_{f,t}). \tag{5.14}$$

From (5.14), we can see that the clauses of regression in (5.8) are true.

3. (Initial state). We need to show that for each variable x in $\mathcal{X}$ such that $s_{\mathcal{I}}(x) = f$:

$$\bigvee_{\forall g, \delta_{f\to g} \in \mathcal{T}} U_{f,g,1} \tag{5.15}$$

According to our construction, (5.15) becomes:

$$\bigvee_{\forall g, \delta_{f\to g} \in \mathcal{T}} (W_{f,1} \wedge W_{g,2}),$$

which is equivalent to:

$$W_{f,1} \wedge \Big(\bigvee_{\forall g, \delta_{f\to g} \in T(x)} W_{g,2} \Big) \tag{5.16}$$

Since f is in the initial state, $\Gamma_\Psi(W_{f,1}) = \Gamma'_\Psi(W_{f,1}) = \top$. Therefore the first part of the conjunction in (5.16) is true. The rest part of (5.16) can be seen to be true following a similar argument as that for the progression case.

4. (Goal). The goal clauses can be shown in a similar way as that for the initial state clauses.

5. (Composition of actions). The clauses we want to prove to be true are, for any action a, $U_{a,t} \rightarrow \bigwedge_{\forall \delta \in M(a)} U_{\delta,t}$, or equivalently, $\overline{U_{a,t}} \vee \bigwedge_{\forall \delta \in M(a)} U_{\delta,t}$.

   Suppose $M(a) = \{\delta_{f_1 \rightarrow g_1}, \delta_{f_2 \rightarrow g_2}, \dots, \delta_{f_m \rightarrow g_m}\}$. The clause we need to show becomes:

$$(\overline{W_{a,t}} \vee W_{f_1,t}) \wedge (\overline{W_{a,t}} \vee W_{g_1,t}) \wedge (\overline{W_{a,t}} \vee W_{f_2,t}) \wedge (\overline{W_{a,t}} \vee W_{g_2,t}) \wedge \dots \\ \wedge(\overline{W_{a,t}} \vee W_{f_m,t}) \wedge (\overline{W_{a,t}} \vee W_{g_m,t}) \qquad (5.17)$$

   Let us call these two-literal disjunctions in (5.17) as sub-clauses. All those $\overline{W_{a,t}} \vee W_{f_i,t}$ sub-clauses in (5.17) are exactly the same as the precondition clause (class IV) in PE. So all $\overline{W_{a,t}} \vee W_{f_i,t}$ in (5.17) are satisfied.

   Next, let us consider those $\overline{W_{a,t}} \vee W_{g_i,t}$ sub-clauses. For any $g = g_i, i = 1, \cdots, m$. There are four cases where $W_{a,t}$ and $W_{g,t}$ are assigned different values:

   - $(W_{a,t} = \perp, W_{g,t} = \perp)$: $\overline{W_{a,t}} \vee W_{g,t}$ is satisfied.
   - $(W_{a,t} = \perp, W_{g,t} = \top)$: $\overline{W_{a,t}} \vee W_{g,t}$ is satisfied.
   - $(W_{a,t} = \top, W_{g,t} = \top)$: $\overline{W_{a,t}} \vee W_{g,t}$ is satisfied.
   - $(W_{a,t} = \top, W_{g,t} = \perp)$: According to Lemma 2, $W_{g,t}$ is a free variable. Therefore, since $\Gamma'_\Psi(W_{g,t}) = \top$, $\overline{W_{a,t}} \vee W_{g,t}$ is satisfied by Γ'_Ψ, and hence satisfied by Γ_Π.

6. (Transition mutex). Consider any mutex clause between two regular transitions $\delta_1 = \delta_{f \rightarrow g}$ and $\delta_2 = \delta_{f' \rightarrow g'}$. Let $\delta_{f \rightarrow g} \in M(a)$ and $\delta_{f' \rightarrow g'} \in M(b)$, we see that a and b are S-mutex. According to Lemma 4, a and b are also P-mutex in PE. Therefore, we have $\overline{W_{a,t}} \vee \overline{W_{b,t}}$. From our construction, we know $\overline{U_{a,t}} \vee \overline{U_{b,t}}$. Then, since we have the composition of actions, $U_{a,t} \rightarrow \bigwedge_{\forall \delta \in M(a)} U_{\delta,t}$ and $U_{b,t} \rightarrow \bigwedge_{\forall \delta \in M(b)} U_{\delta,t}$. A simple resolution of these clauses yields $\overline{U_{\delta_1,t}} \vee \overline{U_{\delta_2,t}}$, which equals to the transition mutex clause $U_{\delta_1,t} \rightarrow \overline{U_{\delta_2,t}}$. Therefore, the transition mutex clause is true in Γ_Π. A similar argument applies when the transitions are prevailing and mechanical.

7. (Action existence). The clauses that we want to prove are $U_{\delta,t} \rightarrow \bigvee_{\delta \in M(a)} U_{a,t}$, for any transitions δ. By our construction, the clauses become

$$\overline{W_{f,t}} \vee \overline{W_{g,t+1}} \vee \bigvee_{\forall a \in A(\delta_{f \rightarrow g})} W_{a,t}. \qquad (5.18)$$

Let $\delta = \delta_{f \to g}$. First, we know by definition that $\bigcup_{\forall h} A(\delta_{h \to g}) = \text{ADD}(g)$. Let us denote $\text{ADD}(g)$ as Z. According to clauses of class III in PE, there are clauses:

$$W_{g,t} \to \bigvee_{\forall a \in Z} W_{a,t}. \tag{5.19}$$

We divide Z into multiple action sets according to different fact from $\{f, h_1, \ldots, h_m\}$, denoted as $Z_f, Z_{h_1}, \cdots, Z_{h_m}$. In fact, for each $h \in \{f, h_1, \ldots, h_m\}$, Z_h is equivalent to $A(\delta_{h \to g})$. Consider any $h_i, i = 1, \cdots, m$. According to the clauses of class IV, for every action $a \in \text{PRE}(h)$, there is a clause $W_{a,t} \to W_{h_i,t}$, which is

$$\overline{W_{a,t}} \vee W_{h_i,t}. \tag{5.20}$$

Next, we perform resolutions by using (5.19) and all the clauses in (5.20), for all such h_i and corresponding actions. We consequently have:

$$\overline{W_{g,t}} \vee (W_{h_1,t} \vee W_{h_2,t} \vee \ldots \vee W_{h_m,t}) \vee \bigvee_{\forall a \in Z_f} W_{a,t}. \tag{5.21}$$

Further, note that all $h_1, h_2, \ldots, f_m$ are mutex to f, a resolution using all the mutex clauses in PEresults in:

$$\frac{(5.21),\ \overline{W_{h_1,t}} \vee \overline{W_{f,t}},\ \overline{W_{h_2,t}} \vee \overline{W_{f,t}},\ \ldots, \overline{W_{h_m,t}} \vee \overline{W_{f,t}}}{\overline{W_{g,t}} \vee (\overline{W_{f,t}} \vee \overline{W_{f,t}} \vee \ldots \vee \overline{W_{f,t}}) \vee \bigvee_{\forall a \in Z_f} W_{a,t}} \tag{5.22}$$

Since $Z_f = A(\delta_{f \to g})$, the outcome of (5.22) leads to (5.18).

8. (Action mutex). Action mutex clauses are satisfied by Γ_Π according to Lemma 4.

Combining all the cases concludes that the constructed solution Γ_Π satisfies all the clauses in SASE which means SASE is satisfiable. Since for all action a, $\Gamma_\Psi(W_{a,t}) = \Gamma'_\Psi(W_{a,t}) = \Gamma_\Pi(U_{a,t})$, Γ_Ψ and Γ_Π represent the same solution plan. □

Theorem 2 *Given a STRIPS task Ψ and a SAS+ task Π that are equivalent, for a time step bound N, if* $\text{SASE}(\Pi, N)$ *is satisfiable,* $\text{PE}(\Psi, N)$ *is also satisfiable.*

Proof Assuming Γ_Π is a satisfiable solution to $\text{SASE}(\Pi, N)$, we first construct an assignment Γ_Ψ from Γ_Π, and show that Γ_Ψ satisfies every clause in $\text{PE}(\Psi, N)$.

Construction. We construct a solution Γ_Ψ as follows:

1. For every $a \in \mathcal{A}$ (which is also in $\mathcal{O}$), we let $W_{a,t} = U_{a,t}$;

2. For every dummy action variable dum_f, we let $W_{dum_f,t} = U_{\delta_{f \to f},t}$;

3. For every transition $\delta_{f \to g} \in \mathcal{T}$, if $U_{x,f,g,t} = \top$ in Γ_Π, we set $W_{f,t} = W_{g,t+1} = \top$ in Γ_Π;

4. For each fact f, if $U_{h,f,t} = \perp$ for every transition $\delta_{h \to f} \in \mathcal{T}$ (which implies that case 3 will not assign a value to f), we set $W_{f,t}$ to be $\perp$.

Satisfiability. Next, we prove that every clause in PE is satisfied by Γ_Ψ. The clauses for the initial and goal states are obviously satisfied. Now we consider the add-effect clauses. The clauses that we want to prove are, for every fact f:

$$W_{f,t} \to \bigvee_{\forall a \in \text{ADD}(f)} W_{a,t-1} \tag{5.23}$$

For a given fact f, we consider all the facts $h \neq f$, such that $\delta_{h \to f} \in \mathcal{T}$. For all such h, there are two further cases:

- There exists a fact h such that $\delta^x_{h \to f} \in \mathcal{T}$ and $U_{x,h,f,t-1} = \top$ in Γ_Π. In the satisfiable SASE instance, the *action existence* clauses in class 6 specify that the truth of a non-prevailing transition δ indicates a disjunction of all actions in $A(\delta)$. Since $U_{x,h,f,t-1} = \top$, it follows that in the SASE instance there is an action $a \in A(\delta^x_{h \to f})$ such that $U_{a,t-1} = \top$. Then, by our construction of Γ_Ψ, we see that both $W_{h,t-1}$ and $W_{f,t}$ are true. Since $W_{f,t}$ and $W_{a,t-1}$, $a \in \text{ADD}(f)$, are all true, (5.23) is satisfied by Γ_Ψ.

- If for every fact h that $\delta^x_{h \to f} \in \mathcal{T}$, $U_{x,h,f,t-1} = \perp$ in Γ_Π, then, according to our construction, $W_{f,t} = \perp$ in Γ_Ψ. Thus, Γ_Ψ satisfies (5.23).

The two cases above conclude that Γ_Ψ satisfies the add effect clauses. Next, we show that Γ_Ψ satisfies the precondition clauses, $W_{a,t} \to W_{f,t}$ (i.e. $\overline{W_{a,t}} \vee W_{f,t}$), for all actions $a \in \mathcal{A}$ and facts $f \in pre(a)$. In SASE, we have clauses of class 6, which are $U_{a,t} \to U_{\delta,t}$, for all actions $a \in \mathcal{O}$ and $\delta \in M(a)$. Let the transition be $\delta^x_{f,g}$, we have $\overline{U_{a,t}} \vee (U_{f,t-1} \wedge U_{g,t-1})$, which implies $\overline{U_{a,t}} \vee U_{f,t-1}$. By our construction, we know $\overline{W_{a,t}} \vee W_{f,t-1}$ is true.

Finally, the mutex clauses are satisfied by Γ_Ψ according to Lemma 4. Combining all the cases concludes that the constructed solution Γ_Ψ satisfies all the clauses in PE which means PE is satisfiable. $\square$

From Theorems 1 and 2, we reach the following conclusion.

Theorem 3 *A classical planning problem is solvable by the* PE *encoding if and only if it is solvable by the* SASE *encoding. Further, for solvable problems, the solution plans found by the two encodings have the same, optimal makespan.*

Remarks. We have proved the equivalence between SAS+ based (SASE) and STRIPS based (PE) encodings. The correctness and optimality of SASE consequently follow, by assuming the correctness of PE encoding used in SatPlan06. Furthermore, we have a few following insights.

- In terms of SAT solutions, there is an epimorphism (a surjective mapping) between the solutions to PE and the solutions to SASE. That is, multiple SAT solutions in PE map to one SAT solution in SASE and every SAT solution in SASE is mapped from at least one SAT solution in PE. This is due to the existence of *free variables* in the PE encoding. One solution in SASE corresponds to a group of solutions in PE with the same assignments to real action variables but different assignments to the free variables.

  There are two types of free variables in PE characterized by Lemmas 1 and 2, respectively. The first type includes certain dummy action variables and the second type includes certain fact variables. For example, if a fact f is added by an action a at level t, the fact variable $W_{f,t}$ can still be false if f is not used by any action in a later level as a precondition. The solution satisfies the SAT instance in PE, although semantically the fact assignment is wrong. However, it does not affect the correctness of PE encoding, since we only extract the assignments of variables for real actions, with the dummy actions and facts discarded. In contrast, in SASE all fact variables are implicitly enforced to take correct values through the assignment of transition variables. As a result, SASE enforces stronger constraints that may lead to more effective constraint propagation in SAT solving.

- In terms of action plans, there is an isomorphism (a bijective one-to-one mapping) between PE and SASE. An action plan is extracted from a SAT solution by using the action variables only (excluding dummy actions and fact variables). The constructions in the proofs to Theorems 1 and 2 give the details of this isomorphism. Note that in the constructions, we change the values of free variables, but leave the real action variables intact.

5.3 SAT Solving Efficiency on Different Encodings

In Section 5.1, we showed that PE and SASE are equivalent in terms of semantics. In this section, we study what makes them different regarding practical problem solving. In particular, we want to understand how PE and SASE would make a SAT solving algorithm behave differently.

Modern SAT solvers, which nowadays all employ many sophisticated techniques, are too complicated to be characterized by simple models. In general, it is difficult to accurately estimate the time that a SAT solver needs to solve a SAT instance. In this section, we provide an explanation of why the SAT encodings from SASE are more efficient for SAT solvers to solve than the SAT encodings from PE, and provide empirical evidence to support this explanation.

We discuss SASE's problem structure and why the widely used SAT solving heuristic VSIDS [88] works better on SASE encodings. The idea in VSIDS is to select those variables that appear frequently in the original and learnt clauses, since they lead to stronger constraint reasoning and space pruning. We show that, if we order all the variables by their VSIDS score, the top-ranked transition variables introduced in SASE have much higher VSIDS scores than those top-ranked action variables. As a result, those top-ranked transition variables are selected more often and provide stronger constraint propagation, speeding up SAT solving. Empirically, we define a significance index of transition variables and show that it has strong correlation with the SAT solving speedup. All the analysis in this section uses SatPlan06 as the baseline.

5.3.1 The VSIDS Heuristic in SAT Solving

The SAT solvers we use are based on the DPLL algorithm [29]. In a DPLL algorithm, a *decision variable* refers to the one selected as the next variable for branching. Once a decision variable is chosen, more variables could be fixed by unit propagation. The ordering of decision variables significantly affects the problem solving efficiency. Most existing complete SAT algorithms use variants of VSIDS heuristic [88] as the variable ordering strategy.

The VSIDS heuristic essentially evaluates a variable by using the Exponential Moving Average (EMA) of the number of times (**frequency**) it appears in all the clauses. This frequency value keeps changing because of the learnt clauses. Therefore, VSIDS uses a smoothing scheme which periodically scales down the scores of all variables by a constant in order to reflect the importance of recent changes in frequencies. The variable that occurs the most frequently usually has a higher value, thus also a higher chance of being chosen as a decision variable. A random decision is made if there is a tie. Thus, variables associated with more recent conflict clauses have higher priorities.

We first consider the frequency only. Then we investigate further by taking the periodic update into consideration.

Given the fact that the frequency is used as the major measurement, VSIDS will be most effective when the difference between variables' frequencies are large and there are some variables with high frequencies. If all variables have the same frequency, then picking decision variables will be purely random. Further, variables with high frequencies are desirable since they lead to stronger constraint propagation.

Therefore, because of the way VSIDS works, we want to enlarge the difference between variable frequencies and make some variables have very high frequencies. SASE's problem structure does what is suggested by this motivation.

The two-level structure, as we discussed in Section 5.1.1, makes transition variables more significant. Transition variables take more responsibilities in terms of constraint propagation as many of them appear frequently in clauses. They consequently are more likely to be chosen by the VSIDS heuristic. Suppose we make a total ordering of all variables by their frequencies from high to low, in the following we give empirical studies to show that top ranked transition variables have much higher chance to be assigned since the top ranked variables consist mostly of transition variables rather than action variables.

In the rest of this section, we will show the significance of transition variables, and explain why they make the search more efficient. In Section 5.3.2, we present a comparison on transition variables versus action variables. In Section 5.3.3, we show how often transition variables are chosen as decision variables, which is a direct evidences of transition variables' significance. Section 5.3.4 shows a strong statical correlation between the speedup in SAT solving and an index measuring the significance of transition variables within the VSDIS heuristic.

5.3.2 Transition Variables versus Action Variables

Let us first formalize how the frequency of variables is measured. Given a SAT instance (V, C), we quantify every individual variable $v \in V$, and define a function $h(v)$ to indicate the frequency of v. That is, $h(v)$ is the number of clauses that v appears in.

We further define *percentile* and *top variable set* to quantify our analysis.

Definition 16 (Percentile and Top Variable Set). *Given a SAT instance (V, C), we sort all variables in V by $h(v)$ from high to low. For an integer p, $0 \leq p \leq 100$, the pth percentile, h^p is the*

lowest value such that p percent of variables in V have an h value below h^p. For the given SAT instance (V, C) and the pth percentile, we define $V^p = \{v \mid v \in V, h(v) \geq h^p\}$ as the **top** $p\%$ **variable set**.

Top variable set is to measure the top ranked variables with the highest h value. We use V_o and V_δ to denote the action variables and transition variables in V, respectively. We also define $V_o^p = V_o \cap V^p$, and similarly $V_\delta^p = V_\delta \cap V^p$. Table 5.1 compares the h values of transition variables and action variables in real SAT instances. This table has two parts. For the first part, we list the average and standard deviation of h values for both transition variables and action variables. The data are collected from the first satisfiable instance of the largest solvable instance in every domain. From the average h value, we can tell that in most domains transition variables occur more frequently than action variables do. We can also see that the standard deviation of transition variables are often large. They are in general not only larger than action variables' standard deviation, but also even larger than the expected value of transition variables. The high frequencies of transition variables, along with large standard deviations, are preferred by the VSIDS heuristic and can aid SAT solving, as we discussed before.

The second part lists the average of h value for transition variables and action variables, in the top p variable set with different values of p: 1%, 2%, 5% and 10%. The difference between V_o^p and V_δ^p is dramatic. In most cases, transition variables dominate the top variable sets, while action variables exist only in very few cases. One exception is the Airport domain. In this domain, although the average h value of all transition variables is smaller than the average h value of action variables, if we look at the top 1% variables, the average h value of transition variables is much higher than the average h value of action variables. Since VSIDS picks the variable with the highest heuristic value, transition variables have higher chances of being picked as decision variable.

The above investigation leads to a speculation: *can we expect further improvements, by using a crafted SAT solver which always branches on transition variables first?* The answer is no. We in fact developed a specific SAT solver that always branches on transition variables first. The result shows that such a solver is less effective than a general SAT solver. The reason is as follows. First, although transition variable *usually* deserve higher priority being decision variables, it is not necessary true for all transitions variables. Some action variables, even though very few, may turn out to be important as well. This in fact can be observed from Table 5.1. Second, as we mentioned above, the occurrence is not all that how VSIDS works. The statistical data we presented in Table 5.1 does not reflect the dynamic changes of heuristic values. Therefore, it is better left to SAT solvers to determine variable ordering by the VSIDS heuristic.

Instances	N	V_δ		V_o		$\overline{h}$ of V_δ^p				$\overline{h}$ of V_o^p			
		$\overline{h}$	σ	$\overline{h}$	σ	1%	2%	5 %	10%	1%	2%	5 %	10%
Airport-48	68	8.6	7.6	19.6	12.9	98.5	75.5	59.2	23.4	39.6	35.9	34.7	32.4
Depot-14	12	10.5	6.0	6.3	3.3	32.5	28.6	23.7	20.9	-	-	-	-
Driverlog-16	18	32.3	11.1	5.6	3.1	43.9	34.6	26.5	23.4	-	-	-	-
Elevator-16	15	22.2	7.4	10.2	3.8	27.0	18.9	18.9	18.9	-	-	-	-
Freecell-6	16	42.6	58.0	33.2	7.0	115.6	86.0	49.0	34.7	-	-	-	-
Openstacks-2	23	14.1	5.2	11.5	4.3	17.2	17.2	16.2	15.2	-	-	-	-
Parcprinter-20	19	12.0	11.8	15.5	5.9	44.3	42.8	26.7	17.8	30.0	30.0	30.0	30.0
Pathways-17	21	5.5	8.5	12.9	3.6	33.6	26.9	15.9	14.5	-	-	-	-
Pegsol-25	25	23.0	15.5	15.2	6.3	30.0	29.8	17.2	15.5	-	-	-	-
Pipe-notankage-49	12	22.9	47.1	41.3	3.4	77.1	57.5	36.4	25.3	-	-	-	-
Pipe-tankage-26	18	58.1	116.7	50.7	12.8	266.4	174.0	86.8	56.0	-	-	-	-
Rovers-18	12	15.5	14.6	16.0	6.9	175.1	86.0	35.5	35.5	-	-	-	-
Satellite-13	13	31.8	7.8	2.0	0.3	35.0	35.0	35.0	35.0	-	-	-	-
Scanalyzer-28	5	113.0	151.4	8.6	1.1	242.8	175.8	129.7	129.7	-	-	-	-
Sokoban-6	35	15.6	4.8	20.0	4.7	16.6	14.1	12.8	11.2	-	-	-	10.0
Storage-13	18	4.7	1.9	6.3	1.6	10.4	10.4	9.0	8.1	-	-	-	9.0
TPP-30	11	12.3	16.1	4.8	0.7	84.2	57.8	34.4	24.6	-	-	-	-
Transport-17	22	22.8	19.0	4.5	1.1	99.6	58.1	52.0	41.6	-	-	-	-
Trucks-13	24	5.1	7.6	6.4	1.2	56.7	38.2	20.8	16.3	-	-	-	-
Woodworking-30	4	6.2	5.1	10.2	3.5	23.1	22.1	18.9	17.2	-	-	-	13.1
Zenotravel-16	7	20.2	25.0	3.9	0.3	51.3	51.3	36.2	28.5	-	-	-	-

Table 5.1: The h values of transition variables versus action variables in all domains. Column 'N' is the optimal makespan. Column '$\overline{h}$' is the average and Column 'σ' is the standard deviation. Column '$\overline{h}$ of V_δ^p' and '$\overline{h}$ of V_o^p' refer to the average h value of transition variables and action variables in V^p, while p equals to 1, 2, 5 or 10. '-' means there is no variable in that percentile range.

5.3.3 Branching Frequency of Transition Variables

In Section 5.3.2, we have seen the difference there is between transition variables and action variables, in terms of the h values. As we mentioned above, however, VSIDS heuristic periodically updates heuristic values of all variables. The dynamic updating of heuristic values is so far not captured by above analysis. In the following, we present more direct empirical evidence to show that transition variables are indeed decided chosen more frequently than action variables for branching, especially at early stages of SAT solving. That is, a SAT solving engine spends most of its time on deciding an appropriate transition plan. This analysis takes into consideration VSIDS's dynamic updating strategy.

We empirically test the probabilities that transition variables and action variables are chosen as branching variables. We measure for every k consecutive decision variables, the number of transition variables (M_δ) and action variables (M_o) that are selected as the decision variables. Let V_δ and

V_o be the sets of transition variables and action variables, respectively, in a SASE encoding. If all variables are selected equally likely, we should have

$$E(M_\delta) = k\frac{|V_\delta|}{|V_\delta| + |V_o|} \text{ and } E(M_o) = k\frac{|V_o|}{|V_\delta| + |V_o|}, \tag{5.24}$$

which implies:

$$\frac{E(M_\delta)}{k|V_\delta|} = \frac{E(M_o)}{k|V_o|} \tag{5.25}$$

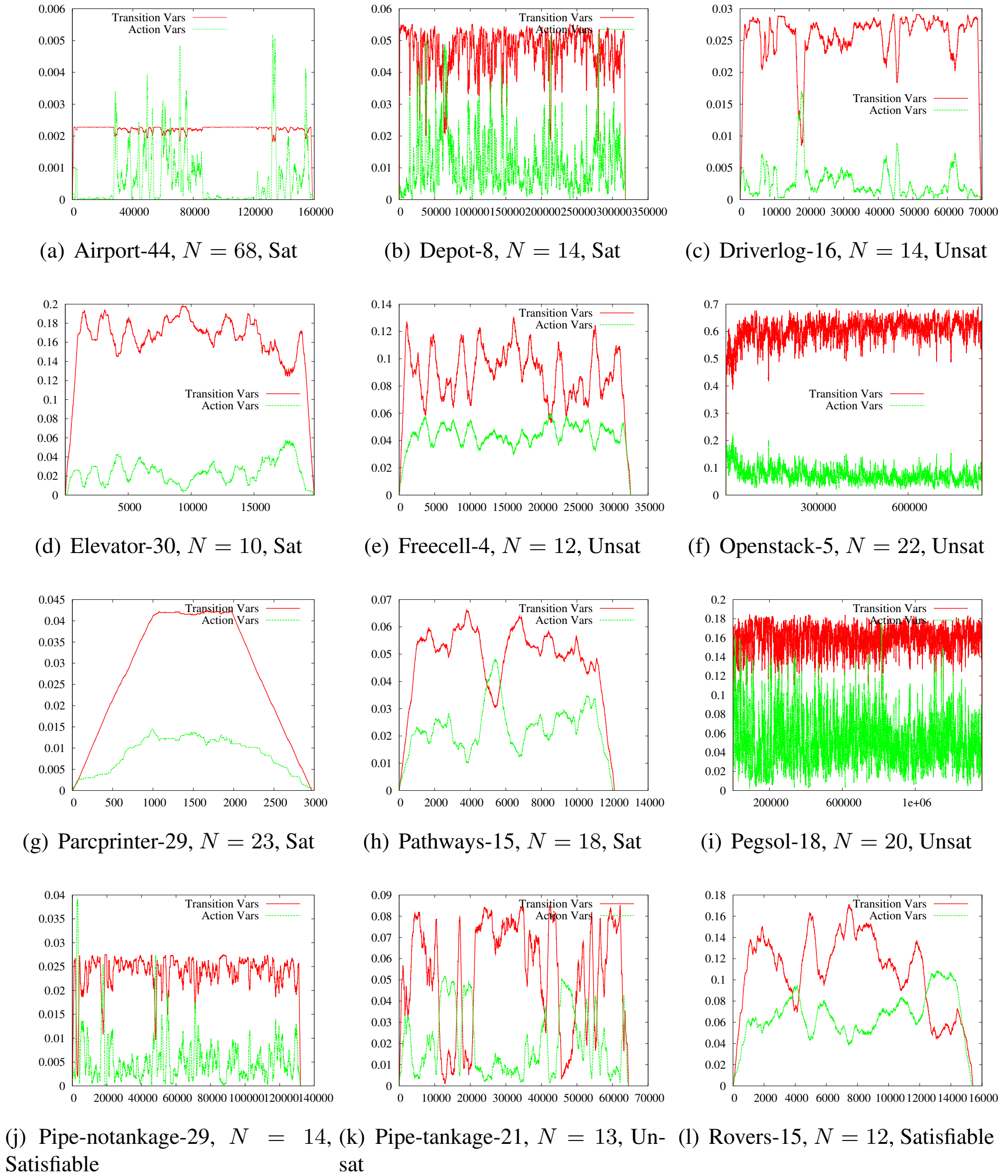
In Figures 5.2 and 5.3 in the Appendix, we show some empirical results where we divide the SAT solving process into epoches of length $k = 1000$ each, for all domains from IPC-3 to IPC-6. In each domain, we choose a instance with at least 500,000 decisions. In some domains (e.g. Woodworking), even the biggest instance has only thousands of decisions. In such a case, we choose the instance with the biggest number of decisions. We also choose the instances to be half UNSAT and half SAT, if we can find appropriately large ones in the particular domain. For every epoch, we plot the **branching frequency**, which is $\frac{M_\delta}{k|V_\delta|}$ for transition variables and $\frac{M_o}{k|V_o|}$ for action variables, respectively. According to (5.25), these two branching frequencies should be about the same if the two classes of variables are equally likely to be chosen.

From Figures 5.2 and 5.3, it is evident that, for all these instances except Storage-12 and Woodworking-20, the branching frequency of transition variables are higher than that of action variables. In fact, in many cases, the branching frequencies of transition variables can be up to 10 times higher than those of action variables. In Transport-26 and Zenotravel-15, the difference is orders of magnitude larger. Such results clearly show that the SAT solving engine branches on the transition variables more frequently than on action variables, which corroborates our analysis in the previous subsection that some of the transition variables introduced by SASE have a higher h values during SAT solving and enhance the strength of VSIDS heuristic.. It shows that the SAT engine spends the majority of time choosing correct transition variables, and when the transition variables are fixed, most action variables can be fixed through constraint propagation instead of branching.

The observation on branching frequency greatly confirms the fact that, statistically, transition variables are picked more often, and assigned values much earlier than action variables.

5.3.4 Transition Index and SAT Solving Speedup

The behaviors of transition variables, as presented above, strongly suggest that there are correlation between the significance of transition variables and the speedup SASE achieves. Nevertheless, the



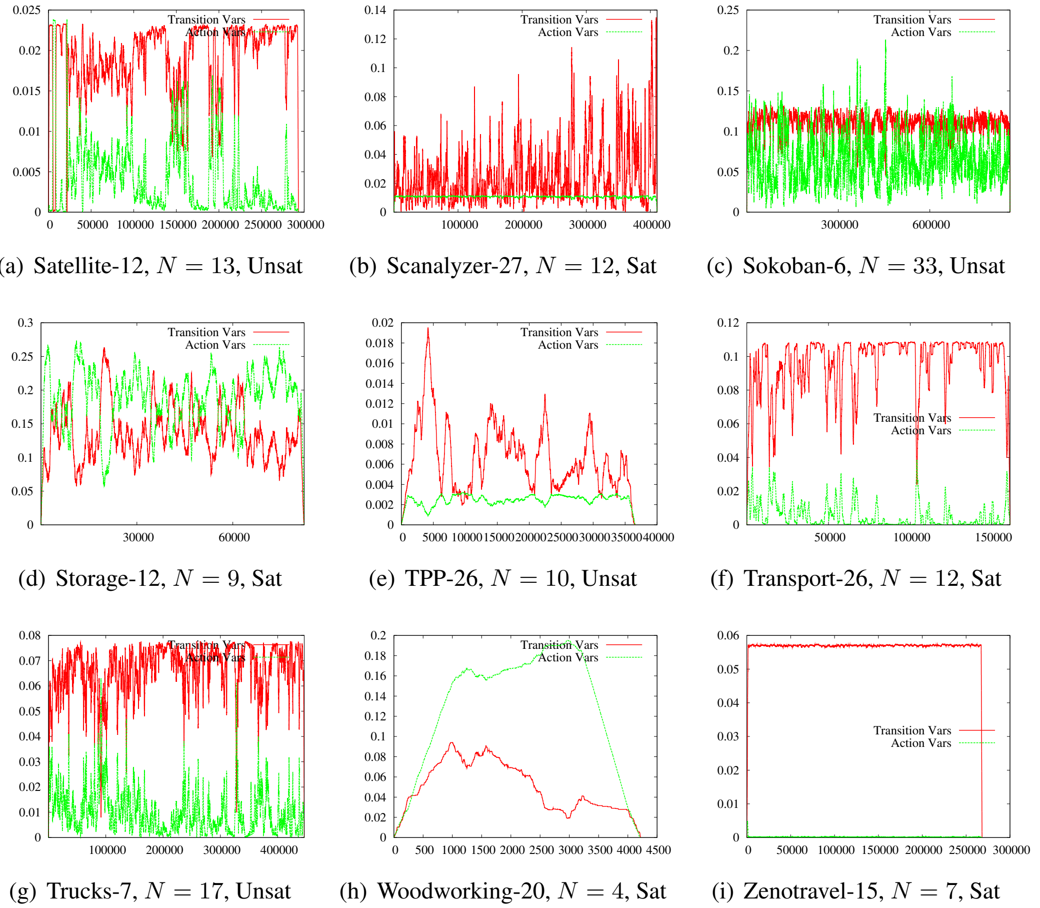
(a) Airport-44, $N = 68$, Sat (b) Depot-8, $N = 14$, Sat (c) Driverlog-16, $N = 14$, Unsat

(d) Elevator-30, $N = 10$, Sat (e) Freecell-4, $N = 12$, Unsat (f) Openstack-5, $N = 22$, Unsat

(g) Parcprinter-29, $N = 23$, Sat (h) Pathways-15, $N = 18$, Sat (i) Pegsol-18, $N = 20$, Unsat

(j) Pipe-notankage-29, $N = 14$, Satisfiable (k) Pipe-tankage-21, $N = 13$, Unsat (l) Rovers-15, $N = 12$, Satisfiable

Figure 5.2: Comparisons of variable branching frequency (with $k = 1000$) for transition and action variables in solving certain SAT instances in twelve benchmark domains encoded by SASE. Each figure corresponds to an individual run of MiniSAT. The x axis corresponds to all the decision epochs during SAT solving. The y axis denotes the branching frequency (defined in the text) in an epoch of $k = 1000$.



(a) Satellite-12, $N = 13$, Unsat

(b) Scanalyzer-27, $N = 12$, Sat

(c) Sokoban-6, $N = 33$, Unsat

(d) Storage-12, $N = 9$, Sat

(e) TPP-26, $N = 10$, Unsat

(f) Transport-26, $N = 12$, Sat

(g) Trucks-7, $N = 17$, Unsat

(h) Woodworking-20, $N = 4$, Sat

(i) Zenotravel-15, $N = 7$, Sat

Figure 5.3: Comparisons of variable branching frequency (with $k = 1000$) for transition and action variables in solving certain SAT instances in nine other benchmark domains encoded by SASE.

study on branching frequency only profiles the connection by showing *what exactly happens during* SAT solving. Another interesting study should reveal *what leads to the speedup* in a more direct way. To quantify the analysis, we define a measurement called the *transition index*.

As mentioned earlier, the h value does not exactly reflect how VSIDS works, as it updates dynamically throughout SAT solving. Nevertheless, by putting together all variables and study their $h()$, the statistics on the population leads to interesting observations. That is the motivation of the transition index, as specified in Definition 17.

Definition 17 (Transition Index). *Given a planning problem's SAT instance (V, C), we measure the top $p(0 \leq p \leq 100)$ variable set, and calculate the transition index of p as follows:*

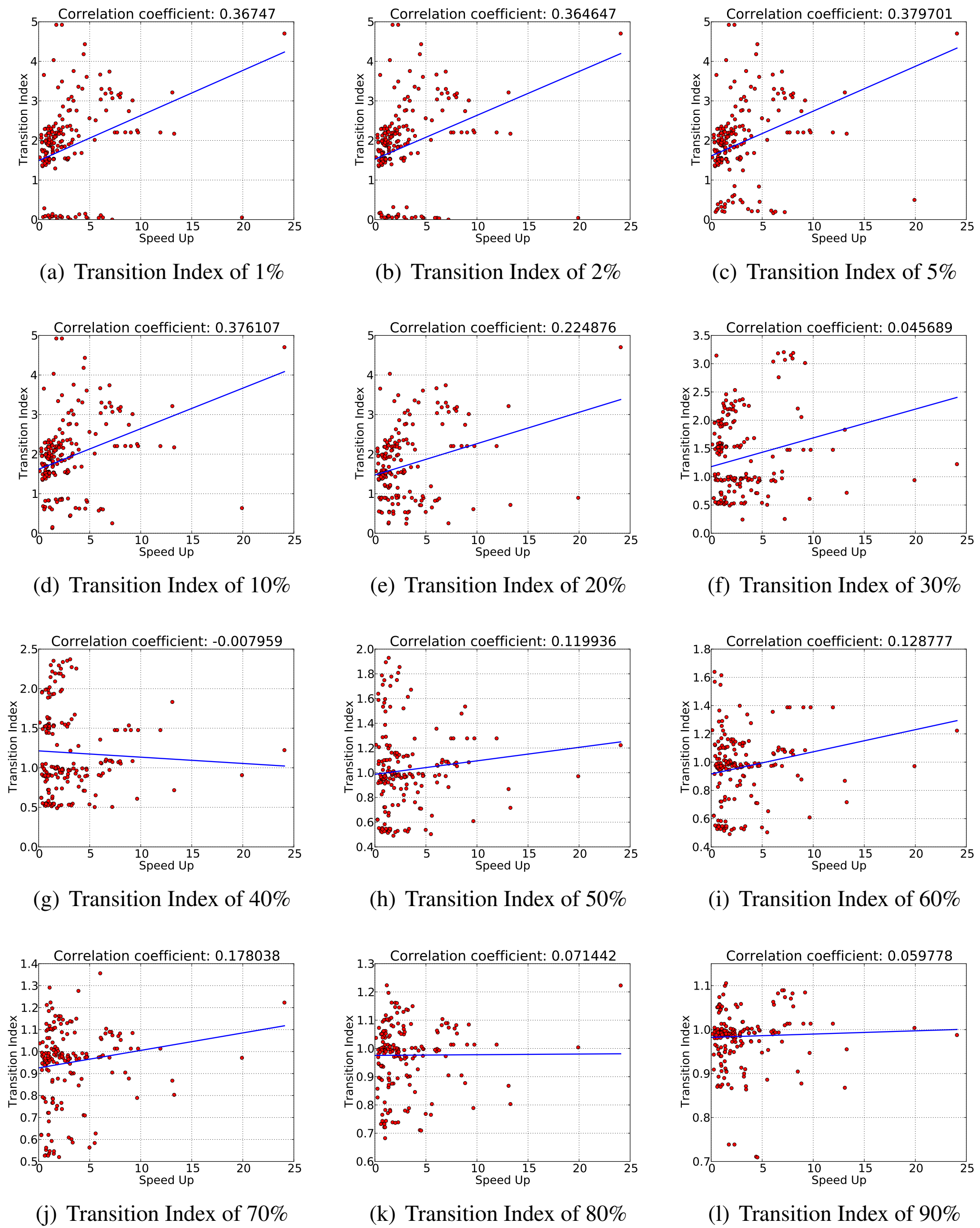
$$\frac{|V_\delta^p|/|V^p|}{|V_\delta|/|V|}$$

Essentially, the transition index measures the relative density of transition variables in the top variable set. If the distribution of the transition variables is homogeneous under the total ordering based on h, $|V_\delta^p|/|V^p|$ should be equal to $|V_\delta|/|V|$ for any given p. A transition index larger than 1 indicates that the transition variables have a higher-than-normal density in the top $p\%$ variable set. The larger the transition index is, the more skewed the density of the transition variables is in the top $p\%$ variable set.

Given a planning problem's SAT instance, there is strong correlation between its transition index and the speedup SASE obtains. In Figure 5.4 we measure such correlation for all the domains from IPC-3 to IPC-6. 12 transition indexes are presented: 1%, 2%, 5%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80% and 90%. Each dot in the figure refers to an individual planning instance. Bootstrap aggregating [13] is used for the regression lines. For each measurement, we calculate Spearman's rank correlation coefficient [90]. It assesses how well the relationship between two variables can be described using a monotonic function. If there are no repeated data values, a perfect Spearman correlation of +1 or -1 occurs when each of the variables is a perfect monotone function of the other.

The instances included in Figure 5.4 are those solved by both SatPlan06 and SASE, with Precosat as the SAT solver. In total we have 186 instances. The speedup of each instance is SASE's SAT solving time divided by SatPlan06's SAT solving time, which is larger than 1 in most cases. To reduce noises, we do not consider those small instances that both SASE and SatPlan06 spend less than 1 second to solve.

We can see that a larger transition index leads to a higher speedup. Variables with higher h values are chosen as decision variables with a higher probability. Thus, a measurement of higher top



(a) Transition Index of 1% (b) Transition Index of 2% (c) Transition Index of 5%

(d) Transition Index of 10% (e) Transition Index of 20% (f) Transition Index of 30%

(g) Transition Index of 40% (h) Transition Index of 50% (i) Transition Index of 60%

(j) Transition Index of 70% (k) Transition Index of 80% (l) Transition Index of 90%

Figure 5.4: The correlation between SAT solving speedup and the transition indexes.

variables is more important, as those low-ranked variables are rarely selected for branching anyway. As expected, the correlation between the transition index and speedup are higher in the transition index with smaller p, such as 1%, 2%, and 5%. Such a result directly links the significance of top ranked (high frequency) transition variables to speedup in SAT solving.

5.4 Reducing the Encoding Size of SASE

We now propose several techniques to further reduce the size of SAT instances in SASE. We first represent all mutual exclusions in SASE using a more compact clique representation. We then develop two new techniques to recognize the special structure of SASE and further reduce the encoding size.

5.4.1 Mutual Exclusion Cliques

A major observation on SASE is that mutual exclusions naturally define cliques of transitions or actions in which at most one of them can be true at any time step. There are two types of cliques: 1) for each $x \in \mathcal{X}$, $\mathcal{T}(x)$ is a *clique of transitions* enforced by the class 5 clauses, and 2) for each transition δ that is not prevailing, $A(\delta)$ is a *clique of actions* enforced by the class 8 clauses.

To encode all mutex within a clique of size n pair-wisely requires $\Theta(n^2)$ clauses. To reduce the number of clauses used, in SASE, we use a compact representation proposed in [104] which uses $\Theta(n \log n)$ auxiliary variables and $\Theta(n \log n)$ clauses. The basic idea is the following. Suppose that we have a clique $\{x, y, z\}$ where at most one variable can be true. we introduce auxiliary variables b_0 and b_1 and clauses $x \Leftrightarrow \overline{b_0} \wedge \overline{b_1}$, $y \Leftrightarrow \overline{b_0} \wedge b_1$ and $z \Leftrightarrow b_0 \wedge \overline{b_1}$.

Note that in PE, mutex are not naturally cliques like in SASE, although it is always possible to do some extra work to explicitly tell PE what could be in cliques. Thus the compact clique representation cannot be effectively applied, unless we mix up two formulism together in a single planner.

5.4.2 Reducing Subsumed Action Cliques

We observe that there exist many action cliques that share common elements, while transition cliques do not have this property. In the following, we discuss the case where one action clique is a subset of another. Given two transitions δ_1 and δ_2, if $A(\delta_1) \subseteq A(\delta_2)$, clique $A(\delta_1)$ is referred to being **subsumed** by clique $A(\delta_2)$.

Instances	Before subsumed		After subsumed	
	count	size	count	size
Pipesworld-20	2548	21.72	516	53.66
Storage-20	1449	12.46	249	60.22
Openstack-10	221	22.44	141	23.4
Airport-20	1024	6.45	604	8.49
Driverslog-15	1848	2.82	1848	2.82

Table 5.2: Statistics of action cliques, before and after the subsumed action cliques are reduced. "count" gives the number of action cliques, and "size" is the average size of the action cliques.

In preprocessing, for each transition $\delta_1 \in \mathcal{T}$, we check if $A(\delta_1)$ is subsumed by another transition δ_2's action clique. If so, we do not encode action clique $A(\delta_1)$. In the special case when $A(\delta_1) = A(\delta_2)$ for two transitions δ_1 and δ_2, we only need to encode one of them.

Table 5.2 presents the number of cliques and their average sizes, before and after reducing action cliques, on some representative problems. The reduction is substantial on most problem domains, except for Driverslog in which no reduction occurred. Note that the average sizes of cliques are increased since smaller ones are subsumed and not encoded.

5.4.3 Reducing Action Variables

Action variables are the majority of all variables. Thus, it is important to reduce the number of action variables. We propose two methods when certain structure of a SAS+ planning task is observed.

Unary transition reduction

Given a transition δ such that $|\mathcal{T}(\delta)| = 1$, we say that the only action a in $\mathcal{T}(\delta)$ is reducible. Since a is the only action supporting δ, they are logically equivalent. For any such action a, we remove $V_{a,t}$ and replace it by $U_{\delta,t}$, for $t = 1, \cdots, N$. An effect of this reduction on a few representative domains can be seen in Table 5.3.

Unary difference set reduction

Besides unary transition variables, an action variable may also be eliminated by two or more transition variables. A frequent pattern is the following: given a transition δ, for all actions in $A(\delta)$, their transition sets often differ by only one transition.

Instances	$\|\mathcal{O}\|$	R$_1$	R$_2$	%
Zeno-15	9420	1800	7620	100.00
Pathway-15	1174	173	810	83.73
Trucks-15	3168	36	300	10.61
Openstack-10	1660	0	400	24.10
Storage-10	846	540	0	63.83

Table 5.3: Number of reducible actions in representative instances. Columns 'R$_1$' and 'R$_2$' give the number of action variables reduced, by unary transition reduction and unary difference set reduction, respectively. Column '%' is the percentage of the actions reduced by both methods combined.

Definition 18 *Given a transition $\delta \in \mathcal{T}$, let $I = \bigcap_{\forall a \in A(\delta)} M(a)$. If for every $a \in A(\delta)$, $|M(a) \setminus I| = 1$, we call the action set $A(\delta)$ a unary difference set.*

Consider a transition δ_1 with $A(\delta_1) = \{a_1, a_2, \ldots, a_n\}$. If $A(\delta_1)$ is a unary difference set, the transition sets must have the following form:

$$M(a_1) = \{\delta_1, \delta_2, \ldots, \delta_k, \theta_1\}$$

$$M(a_2) = \{\delta_1, \delta_2, \ldots, \delta_k, \theta_2\}$$

$$\vdots$$

$$M(a_n) = \{\delta_1, \delta_2, \ldots, \delta_k, \theta_n\}$$

In this case, we eliminate the action variables for $a_1, \cdots, a_n$ by introducing the following clauses. For each $i, i = 1, \cdots, n$, we replace $V_{a_i,t}$ by $U_{\delta_1,t} \wedge U_{\theta_i,t}$, for $t = 1, \cdots, N$. Hence, the action variables can be eliminated and represented by only two transition variables.

Table 5.3 shows the number of reducible actions in several representative problems. In Zenotravel, all action variables can be eliminated when the two reduction methods are used. In Openstack and Storage, there is only one type of reduction that can be applied.

Figure 5.5 shows the number of solvable problems from all the problems in IPC-3 to IPC-6, with increasing limits on running time, memory consumption, number of variables or number of clauses. Precosat is used for all planners. Running time is measured by all the time including preprocessing and problem solving. Memory usage is based on the status report by Precosat. Under the maximum CPU time (1800s) and memory limit (4Gb), when both optimizations are off, SASE solves 397 instances. By turning on either the clique representation or the reduction method, SASE solves 416 and 405 instances, respectively. While both clique and reduction are turned on, SASE solves 426 instances.

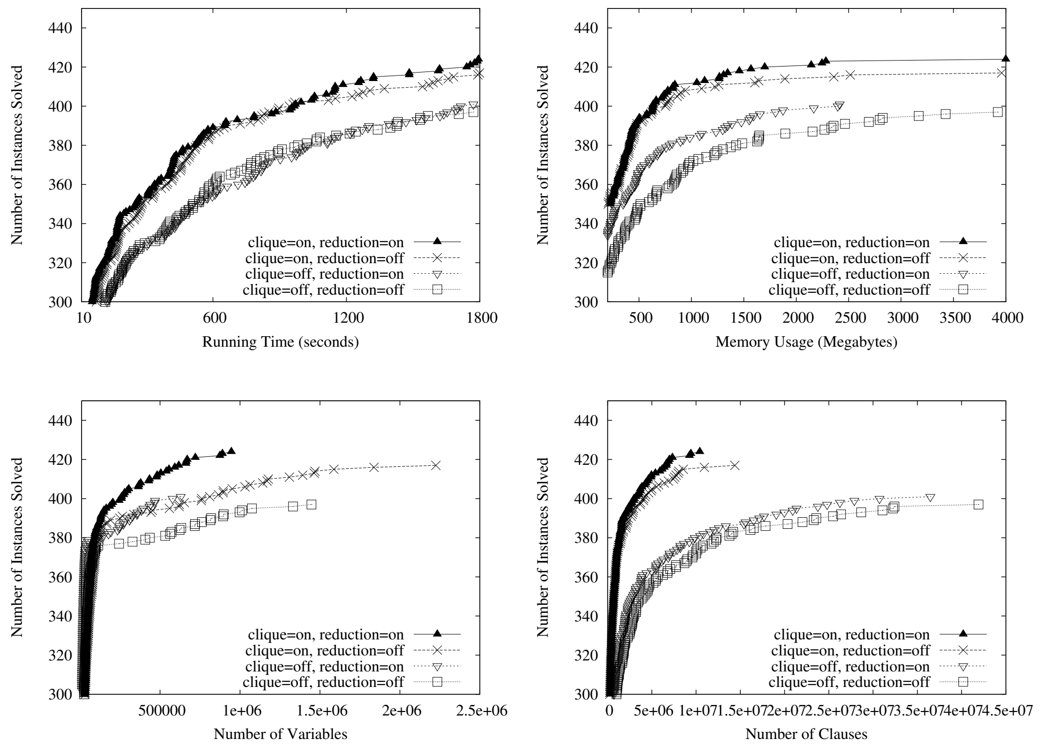


Figure 5.5: The results of SASE while different reduction methods are turned on or off.

The reduction method gives us a few improvements in problem solving time, and the clique representation makes more improvements. For memory consumptions, reduction methods gives moderate amount of improvements, while the clique representation leads to more substantial improvements. For numbers of clauses, the clique technique gives significant reduction. As to numbers of variables, the clique technique gives some improvements, while the reduction method makes some further improvements.

5.5 Experimental Results

We run all experiments on a PC workstation with a 2.3 GHz AMD Quad-Core Opteron processor. The running time for each instance is set to 1800 seconds, and the memory is limited to 4GB. For all planners, the running time includes parsing, preprocessing and problem solving. The memory consumption is the peak memory usage reported by the SAT solvers.

We test all problem instances of STRIPS domains in IPC-3 to IPC-6. PSR and Philosophers are not included because they have derived facts, which cannot be handled correctly by any of the planners tested. We use the parser by Fast-Downward [54, 55] to generate the SAS+ formalism from STRIPS

inputs. The preprocessing and encoding parts of SASE are implemented in Python2.6. All the instances are based on grounded STRIPS.

Precosat (build236) [8], the winner of the application track in the SAT'09 competition, is used as the SAT solver for most planners that we tested and compared. Besides Precosat, we also use CryptoMinisat [119], the winner of SAT Race 2010, as the underlying solver of SatPlan06 and SASE. The nine planners considered are listed as follows.

1. **SP06** and **SP06-Crypto**. They are the original SatPlan06 planner, only with the underlying SAT solver changed to Precosat and CryptoMinisat, respectively.
2. **SASE** and **SASE-Crypto**. They are SASE encoding introduced in this chapter, with all the optimization methods turned on. The underlying SAT solvers are Precosat and CryptoMinisat, respectively.
3. **SP06L**. It is SatPlan06 with long-distance mutual exclusion (londex) [20]. We compare against londex since it also derives transition information from the SAS+ formalism. Here we use domain transition graph from Fast-Downward's parser to derive londex information.
4. **SP06C**. It is SatPlan06 with the clique technique [104] to represent the mutual exclusions. The clique information is obtained via Fast-Downward. Note that due to the different grounding strategies by SatPlan06 and Fast-Downward, not all of the mutual exclusions defined in SatPlan06 can be covered by cliques.
5. **nplan**. The nplan solver is set to use ∀-step to generate plans with the same optimality metric as other planners. The build-in SAT solver is changed to Precosat.
6. **SplitE**. It is the split encoding introduced by Robinson et al. [109] using Precosat.
7. **LM-cut**. This is a sequential optimal planner, using LM-Cut heuristic [56] and A* search. We use the implementation in Fast-Downward.

In Figure 5.6, we present the number of instances that are solvable in the testing domains, with respect to the given time limit and memory limit. It is easy to see that SASE and SASE-Crypto have clear advantages. LM-cut is the least efficient, but again, the comparisons to LM-cut is like comparing oranges and apple, since different optimization metrics are used. For running time, the differences between most other planners are minor, except for SplitE is worse than all other SAT-based planners. For memory consumption, SASE and SASE-Crypto are clearly much better than all other planners. nplan is slightly better than others in smaller instances, but when it comes to larger instances, SatPlan06 becomes more competitive.

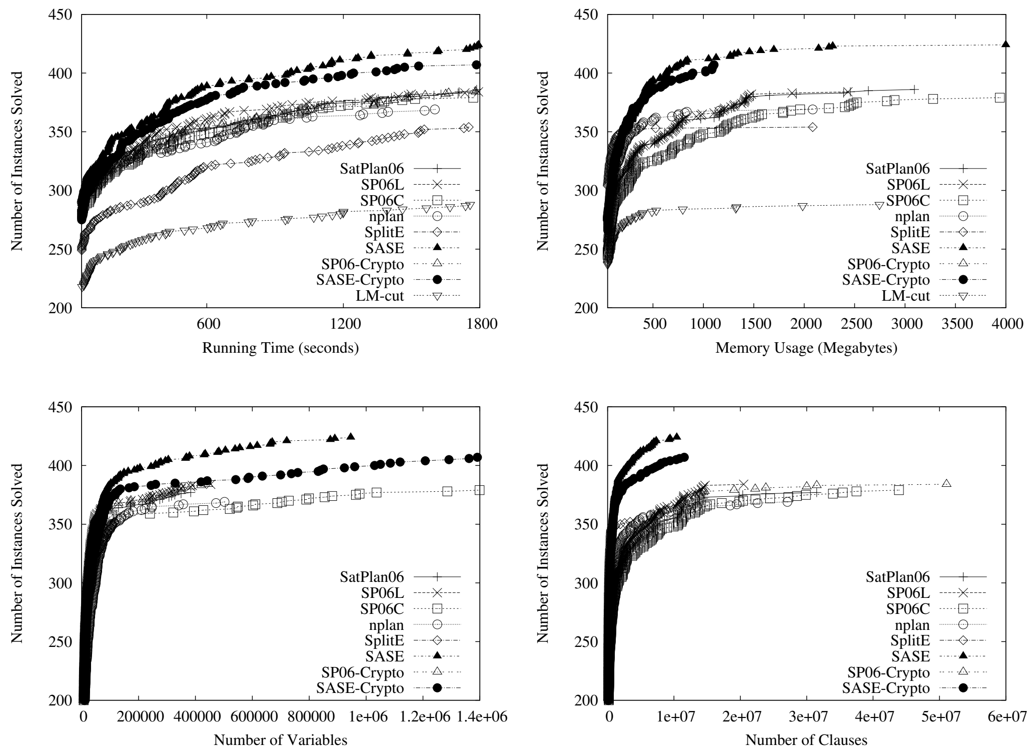


Figure 5.6: Number of problems solved by each planner, with increasing limits on running time, memory consumption, number of variables and number of clauses.

In Figure 5.6, for SAT based planners, we also present the number of instances that are solvable with increasing limits on the number of variables and number of clauses. Note that the curves are slightly affected by the given time and memory limit, thus the same encoding using different SAT solvers may look slightly different. We can however still see SASE's advantages in terms of number of variables. As to number of clauses, SASE is significantly better than other planners.

Table 5.4 presents the number of instances solved in each planning domain, within the given time and memory limit. In general, SASE solves more instances than other planners. nplan has a few bugs that it cannot find the correct solution with the optimal makespan in domains Openstacks, Rovers and Storage. SplitE has bugs in its parser that it cannot handle problems in both Airport and Pathways. Although LM-cut overall solves fewer instances, in a few domains it is much better than all other SAT based planners.

Both SP06L and SP06C use Fast-Downward's parser to get domain transition graph information. As we use STRIPS input for all domains, in some cases Fast-Downward may spent too much time in pre-processing grounded STRIPS instances. That is why the efficiency of londex or clique representation may not compensate the time spent in pre-processing, leading to slightly worse performance than the original SP06 in a few instances. For example, londex is helpful in TPP, but not in Trucks

Domain	SP06	SP06L	SP06C	nplan	SplitE	SASE	SP06^c	SASE^c	LM-cut
Airport	35	38	39	20	0	**46**	38	42	27
Depot	17	16	16	**19**	17	17	17	15	7
Driverlog	16	16	16	**17**	**17**	**17**	**17**	**17**	13
Elevator	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	19
Freecell	5	4	5	**6**	5	**6**	4	**6**	5
Openstacks	5	5	5	0	5	5	5	5	**20**
Parcprinter	29	29	29	**30**	29	**30**	29	**30**	21
Pathways	11	11	11	**12**	0	**12**	9	10	5
Pegsol	21	21	21	21	22	24	18	19	**27**
Pipe-notankage	38	37	31	**40**	37	37	38	35	17
Pipe-tankage	16	16	16	22	10	**26**	13	23	11
Rovers	13	13	13	0	**18**	14	13	17	7
Satellite	17	17	17	**18**	16	**18**	17	17	7
Scanalyzer	15	14	14	**18**	13	**18**	16	17	7
Sokoban	5	5	3	11	5	5	5	5	**24**
Storage	15	15	15	0	**16**	15	15	15	15
TPP	27	**30**	29	28	25	**30**	28	29	6
Transport	19	16	19	**22**	18	**22**	19	21	12
Trucks	7	6	5	**10**	8	8	7	8	**10**
Woodworking	**30**	**30**	**30**	**30**	**30**	**30**	**30**	**30**	16
Zenotravel	15	15	15	15	15	**16**	**16**	**16**	12
Total	386	384	379	369	336	**426**	384	407	288

Table 5.4: Number of instances solved in each domain within 1800 seconds. SP06^c and SASE^c are short for SP06-Crypto and SASE-Crypto.

and Scanalyzer. The clique representation is very helpful in Airport domain, with 10 more instances solved, but does not help too much in Pegsol and Satellite.

Comparing with nplan, in general SASE is better. A quick observation over the comparisons between nplan and SASE is that, nplan performs better than SASE on those domains with very few concurrencies. For example, both Sokoban and Trucks have only one action at nearly each time step. We believe the reason should be the way nplan encodes all mutual exclusions as linear encoding [107]. However, the linear encoding seems not as space efficient as SASE combined with the clique representation.

SplitE in general is slightly worse than SP06. It wins over SP06 on 5 domains and SP06 wins over SplitE on 6 domains. Overall, SplitE is competitive to neither nplan nor SASE, although Rovers is the single domain where SplitE is better than all others.

Although CryptoMinisat won over Precosat in SAT Race 2010, it is not as good for planning problems. For SP06, Precosat solves 386 instances and CryptoMinisat solves 384 instances. For SASE,

Precosat solves 426 and CryptoMinisat solves 407. In both cases, CryptoMinisat leads to fewer instances solved.

In Tables 5.5 and 5.6, we give more details on some of the instances considered. We compare the four fundamentally different encodings that we tested, which are SatPlan06, nplan, SplitE and SASE. We list two largest solvable instances in each domain. If both of the two largest instances are solved by only one planner, we add one more instance which is solved by at least one more planner. We do not include any instance that all planners spend less than 100 seconds to solve. For example, all instances in the Woodworking domain take all planners less than 100 seconds, thus are not included in this table. Usually a less memory consumption (which includes the memory consumption during the problem solving) leads to faster problem solving. In some cases, having less variables and clauses does not necessarily result in less memory consumption. Although Figure 5.6 shows that SASE is in general more memory efficient, in some cases nplan or SplitE may have smaller memory consumptions.

5.6 Summary

We have devised a new encoding scheme (called SASE) that compiles SAS+ representation directly into a SAT formula [65]. In a typical SAT formula that is compiled from STRIPS, the constraints are organized in a strongly coupled manner. SASE models a planning problem with a search space consisted of two hierarchical subspaces. The top subspace is the space of transition plans, and the lower subspace is the space of supporting action plans corresponding to feasible transition plans. We have proved that SASE enforces the same semantics to SatPlan06. We have also conducted empirical study to explain the reason why SASE is more favorable to modern SAT solvers. The experiments on IPC domains show that SASE has clear advantage over all state-of-the-art SAT-based planners.

Instances	SatPlan06				nplan				SplitE				SASE			
	Time	Mem	Var	Clause	Time	Mem	Var	Clause	Time	Mem	Var	Clause	Time	Mem	Var	Clause
Airport-30	1345.5	555	316026	5009883	TLE				TLE				421.0	400	505124	4501651
Airport-32	TLE				TLE				TLE				1056.8	2143	874243	9412635
Airport-46	TLE				TLE				TLE				965.8	663	720910	6881567
Depot-9	TLE				743.0	190	54906	637624	TLE				TLE			
Depot-11	1629.1	515	28444	3377133	239.2	55	45414	488004	1091.3	78	65142	304538	TLE			
Depot-15	TLE				868.6	177	106184	1231785	TLE				TLE			
Driverlog-16	TLE				633.3	211	100505	1385274	1446.3	92	95830	334263	945.5	1126	102813	1041842
Driverlog-17	902.1	517	61915	2752787	263.1	177	89450	1429188	813.3	95	75567	279720	354.0	767	82289	889283
Freecell-5	14.5	11	14201	55575	424.9	312	56012	4579440	1149.7	62	23894	241396	971.0	316	60885	1013019
Freecell-6	TLE				317.9	505	76609	7717043	TLE				168.8	153	69953	1257834
Openstack-2	100.6	19	3709	66712	-				34.6	14	2926	49411	17.8	7	5610	23725
Parcprinter-30	TLE				1602.0	89	24154	1132558	TLE				3.1	15	25388	135760
Pathways-8	TLE				1273.3	84	20908	351383	-				TLE			
Pathways-13	1081.8	85	19267	597022	849.7	115	26366	521023	-				1264.7	94	27411	158557
pegsol-21	TLE				TLE				1720.1	70	16451	86148	1148.7	125	12684	95230
Pegsol-23	TLE				TLE				TLE				1320.9	137	13419	100832
Pegsol-25	TLE				TLE				TLE				1149.5	147	14713	113359
Pipe-notank-30	TLE				1038.4	398	75046	855694	TLE				TLE			
Pipe-notank-33	723.2	790	31366	5867650	770.7	186	51594	662086	1133.8	147	86246	327837	TLE			

Table 5.5: Detailed results on various of instances. Column ‘Time’ is the total running time. Columns ‘Var’, ‘Clause’, ‘Mem’ are the number of variables, number of clauses and memory consumption (in Megabytes), respectively, of the largest SAT encoding. ‘TLE’ is short for memory limit exceeded and a ‘-’ indicates the planner fails to solve the instance.

Instances	SatPlan06				nplan				SplitE				SASE			
	Time	Mem	Var	Clause	Time	Mem	Var	Clause	Time	Mem	Var	Clause	Time	Mem	Var	Clause
Pipe-tank-27	TLE				1370.7	2084	246490	27094158	TLE				658.6	573	185045	4625067
Pipe-tank-29	TLE				TLE				TLE				1254.9	741	259834	6988262
Pipe-tank-35	TLE				TLE				TLE				1788.3	823	272595	7326119
Rovers-13	TLE				-				52.9	16	12759	76862	TLE			
Rovers-17	TLE				-				90.4	28	21146	99847	TLE			
Rovers-18	TLE				TLE				50.7	26	24394	122364	402.0	79	32436	313680
Satellite-12	375.1	203	53649	1734497	108.8	55	53895	380954	1422.8	17	13931	109405	785.2	2283	55534	628510
Satellite-16	TLE				TLE				TLE				1482.3	1699	67707	800710
Scanalyzer-27	TLE				895.5	388	246312	5277880	TLE				247.5	348	66615	2341387
Scanalyzer-28	TLE				TLE				TLE				252.4	378	84029	3880659
Sokoban-4	TLE				872.4	190	58346	1352942	TLE				TLE			
Sokoban-5	489.7	306	24298	1111364	29.3	50	31768	418226	867.3	156	58182	397866	526.1	713	36207	509707
Sokoban-9	TLE				1557.3	248	40314	493782	TLE				TLE			
Storage-13	92.3	84	7369	362345	-				24.0	28	13114	69982	143.9	107	11986	128903
Storage-16	TLE				-				704.9	64	35789	115952	TLE			
TPP-28	TLE				TLE				TLE				1322.2	361	128724	961437
TPP-30	TLE				242.0	817	502224	19462064	TLE				1051.0	341	136106	997177
Transport-8	TLE				964.4	181	104226	1459257	TLE				1109.7	465	100671	936211
Transport-17	TLE				1396.3	201	133381	1842503	TLE				1145.4	630	112506	1057854
Trucks-9	TLE				309.2	134	70296	1413044	942.1	358	50712	2586756	TLE			
Trucks-13	TLE				1064.5	384	166090	4822819	TLE				1776.1	257	205012	1824292
Zenotravel-14	157.9	689	26201	6632923	17.1	62	42445	831247	265.9	13	8019	73165	27.5	42	37620	344856
Zenotravel-15	419.1	974	33259	8956087	62.3	132	73360	1935689	558.8	29	14373	99208	142.7	146	46788	506540

Table 5.6: Detailed results on various of instances on IPC benchmark domains.

Chapter 6

Temporally Expressive Planning as Satisfiability

In this chapter, we extend the planning as SAT approach to tackle temporal planning. Temporal planning is important as temporal constraints are inherent in most planning problems. Temporal planning is also difficult and much more complex than classical planning. Despite that temporal planning is important and much effort has been devoted to it, most existing temporal planners, including SGPlan [129] and CPT [128], do not support temporally expressive planning [26]. A planning task is temporally expressive if it has *required concurrency* property (otherwise it is temporally simple), which is supported by PDDL2.1 semantics. A problem has a required concurrency if there exists a plan for solving the problem and every solution has concurrently executed actions.

Most existing planners make some assumptions on how actions interact with one another. The only existing PDDL based temporally expressive planner that we are aware of is Crikey [24, 23]. Crikey combines planning and scheduling for temporal problems, and uses state-based forward heuristic search, which is Enforced Hill Climbing (EHC) followed by Best-First Search if EHC fails. Nevertheless, Crikey is still not complete with regarding to temporal expressiveness. On the other hand, Cushing et al. also points out that all the temporal planning benchmarks in recently planning competitions are not temporally expressive [26]. In other words, those existing temporal planning problems have no essential difference to classical planning problems.

The lack of both temporally expressive planners and benchmarks is in sharp contrast with the reality that many real-world planning problems are highly concurrent. Inspired by the enormous success of the SAT-based planning paradigm, we adopt its basic idea to formulate temporally expressive planning problems. Our work is also in part inspired by the studies on the advantages of applying SAT testing to general temporal problems [95].

We follow the temporal planning model in PDDL2.1 specification [41] with a few assumptions: each action is grounded, and of constant duration. In addition, we assume time to be discrete.

Although PDDL2.1's semantics have certain limitations thus may not be expressive enough for certain scenarios [2], it is so far the most widely accepted. Most existing temporal planners adopt PDDL2.1 semantics.

Similar to STRIPS, in PDDL2.1 a state of the world is determined by a set of propositions. An action is defined by 'conditions' and 'effects', which are essentially propositions but with temporal information enforced. A condition may be either of following three types: 'at start', 'at end' or 'overall'. For an 'at start' or 'at end' condition, the corresponding proposition needs to be true at the beginning or the ending time point of the action. As to 'overall' condition, the corresponding proposition needs to be true throughout the action's life time. An action is applicable to a given state, only if *all* its conditions are satisfied. Effects may be either 'at start' or 'at end', indicating the point the event occurs.

This chapter is organized as follows. First in Section 6.1 we formalize temporal planning in a STRIPS style. Then we introduce an encoding that handles temporal planning tasks based on this formulations in Section 6.2. The second part is to apply SASE encoding strategy to temporal planning. We first present a formulation of temporal planning in a SAS+ style in Section 6.3, and present the corresponding encoding scheme in Section 6.4. Finally, we discuss the existing temporally expressive domains and conduct experiments by comparing our two encodings to the existing search based temporally expressive planners.

6.1 Temporally Expressive Planning

In this dissertation, we consider temporal planning in a discrete time space, thus a temporal planning task can be also sketched in a state space fashion just like what is defined for classical planning in Chapter 2. A **fact** f is an atomic proposition that can be either true or false; we use f_t to represent the fact f at time t. A state φ is a set of facts that are true. We use φ_t to represent the state at time t.

Definition 19 (Durative Action). *A durative action a is defined by a tuple*

$$(\rho, \pi_{\vdash}, \pi_{\leftrightarrow}, \pi_{\dashv}, \alpha_{\vdash}, \alpha_{\dashv}),$$

where ρ is the duration of o, respectively; $\pi_{\vdash}, \pi_{\dashv}$ are condition fact sets that must be true at the start or at the end of o, respectively; $\pi_{\leftrightarrow}$ is the overall fact sets that must be true over lifetime, respectively; and $\alpha_{\vdash}, \alpha_{\dashv}$ are the effects at the start and the end of o, respectively.

We assume that action durations and costs are integers where $\rho(o) > 0$ and $\mu(o) \geq 0$. Given a durative action o, we use $\pi_{\vdash}$ to represent $\pi_{\vdash}(o)$. The same abbreviation applies to $\pi_{\leftrightarrow}, \pi_{\dashv}, \alpha_{\vdash}$, and $\alpha_{\dashv}$. In PDDL2.1, the annotations of precondition and overall facts are: 1) $\pi_{\vdash}$: "(at start f)", 2) $\pi_{\dashv}$: "(at end f)", and 3) $\pi_{\leftrightarrow}$: "(over all f)". The annotations of effects are: 1) $\alpha_{\vdash}$: "(at start f)" and 2) $\alpha_{\dashv}$: "(at end f)".

Given a durative action o and a sequence of states $\varphi_t, \varphi_{t+1}, ..., \varphi_{t+\rho(o)-1}$, o is ***applicable*** at time t (denoted as o_t) if the following conditions are satisfied: a) $\forall f \in \pi_{\vdash}, f_t \in \varphi_t$; b) $\forall f \in \pi_{\dashv}, f_{t+\rho(o)-1} \in \varphi_{t+\rho(o)-1}$; and c) $\forall f \in \pi_{\leftrightarrow}, t' \in (t, t+\rho(o)-1), f_{t'} \in \varphi_{t'}$.

Action o's execution at time t will affects states φ_{t+1} and $\varphi_{t+\rho(o)}$. States φ_{t+1} and $\varphi_{t+\rho(o)}$ satisfy o_t's effects if: 1) for each add-effect $f \in \alpha_{\vdash}$, $f_{t+1} \in \varphi_{t+1}$, 2) for each delete-effect $(not\ f) \in \alpha_{\vdash}$, $f_{t+1} \notin \varphi_{t+1}$, 3) for each add-effect $f \in \alpha_{\dashv}$, $f_{t+\rho(o)} \in \varphi_{t+\rho(o)}$, and 4) for each delete-effect $(not\ f) \in \alpha_{\dashv}$, $f_{t+\rho(o)} \notin \varphi_{t+\rho(o)}$.

A temporal planning task Ψ is defined by tuple $(\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$, while $\mathcal{A}$ here is a set of durative actions and $\mathcal{F}$ is a set of propositional facts. For simplicity, we assume every action $o \in \mathcal{A}$ to be *durative* and *grounded*.

Definition 20 (Temporal Plan). *Given a temporal planning task $\Psi = (\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$, a plan $P = (p_0, p_1, \ldots, p_{n-1})$ is a sequence of action sets, where each action set $p_t \subseteq \mathcal{A}$ indicates the actions executed at time t. P is a solution plan if there exists a state sequence $S_0, S_1, ..., S_n$ satisfying: a) $S_0 = \varphi_{\mathcal{I}}$; b) for each action $o_t \in p_t$, o_t is applicable at time t, and S_{t+1}, $S_{t+\rho(o)}$ satisfy o_t's effects; c) for all $f \in \varphi_{\mathcal{G}}$, $f_n \in S_n$.*

The semantics of temporal planning that we consider is as expressive as what is defined in the PDDL2.1 standard [41], except for the discrete time setting. While many research has the issue of being not as expressive as PDDL2.1 specification, our approach is capable of handling temporal expressiveness.

Definition 21 (Required Concurrency). *A temporal planning task Ψ has required concurrency if it has at least one solution plan and* ***every*** *solution of Ψ has concurrently executed actions.*

Definition 22 (Temporal Dependency). *Given two durative actions a and a', we define that a temporally depends on a' when one of the following conditions holds:*

1. *$\exists f \in \pi_{\vdash}(a)$, such that $f \in \alpha_{\vdash}(a')$ and $\neg f \in \alpha_{\dashv}(a')$;*

2. *$\exists \neg f \in \pi_{\vdash}(a)$, such that $\neg f \in \alpha_{\vdash}(a')$ and $f \in \alpha_{\dashv}(a')$.*

Two factors lead to concurrencies in a temporally expressive problem. One is the required concurrent interaction (i.e., concurrent execution) among actions, and the other is enforced deadlines [24]. We do not have any assumption and our approach is capable of handling general required concurrencies.

6.2 A STRIPS Style Encoding Scheme

Given a temporally expressive planning task Ψ, we first compile Ψ into a classical planning task Ψ', along with extra constraints indicating the concurrency information. Then, we follow the typical SAT-based planning method: convert Ψ' and extra constraints into a SAT formula, and call a SAT solver to solve it. Repeat this operation iteratively, until a solution is found.

The compilation from Ψ to Ψ' works as follows, each durative action o is converted into two simple actions plus one propositional fact, written as tuple $(o_{\vdash}, o_{\dashv}, f^o)$. These two simple actions indicate the starting and ending event of o. The fact f^a, when it is true, indicates that o is executing. The idea of transforming durative actions is proposed in [84]. It has several advantages. For example, some techniques from classical planning can be applied without sacrificing the completeness. To distinguish temporal actions and simple actions, we use a, b, c to denote simple actions and o, p, q for durative actions.

We extend the encoding of propositional planning in planning graph to temporal planning using the above transformation. Given a time span N and a simplified temporal planning task $(\mathcal{F}^s, \mathcal{A}^s, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$, we define the following variables for the encoding.

1. Action variables $W_{a,t}$, $0 \leq t \leq N, a \in \mathcal{A}^s$.
2. Fact variables $W_{f,t}$, $0 \leq t \leq N, f \in \mathcal{F}^s$.

We also need the following clauses for the encoding.

(I). Initial state (for all $f \in \varphi_{\mathcal{I}}$): $W_{f,0}$

(II). Goal states (for all $f \in \varphi_G$): $W_{f,t}$

(III). Preconditions of simple actions (for all $a \in \mathcal{A}^s$, $0 \leq t < N$):

$$W_{a,t} \rightarrow \bigwedge_{f \in pre(a)} x_{f,t}$$

(IV). Add effects of simple actions (for all $f \in \mathcal{F}^s$, $0 < t \leq N$):

$$W_{f,t} \rightarrow \bigvee_{\{a | f \in add(a)\}} W_{a,t-1}$$

(V). Durative actions ($\forall o, t,\ o \in \mathcal{A}$, $0 \leq t < t + \rho < N$):

$$W_{o_{\vdash},t} \leftrightarrow W_{o_{\dashv},t+\rho-1}$$

$$W_{o_{\vdash},t} \rightarrow \bigwedge_{t+1 \leq t' \leq t+\rho-1} (W_{f^o,t'})$$

$$W_{o_{\vdash},t} \rightarrow \bigwedge_{t+1 \leq t' \leq t+\rho-1} \big(\bigwedge_{f \in \pi_{\leftrightarrow}} W_{f,t'} \big)$$

If a start action $o_{\vdash}$ is true at time t, then action $o_{\dashv}$ must be true at time $t + \rho - 1$, and vice versa. If a start action $o_{\vdash}$ is true at time t, then the fact f^o and all the overall facts determined by $\pi_{\leftrightarrow}$ must be true in the executing duration $[t + 1, t + \rho - 1]$. These constraints enforce that o is executed in $[t, t + \rho)$. Note it is not necessary to encode this type of constraints for those actions whose duration ρ is smaller than or equal to 1.

(VI). Action mutex ($0 \leq t < N$): for each pair of mutex actions (a_1, a_2):

$$\neg W_{a_1,t} \bigvee \neg W_{a_2,t}$$

(VII). Fact mutex ($0 \leq t \leq N$): for each pair of mutex facts (f_1, f_2):

$$\neg W_{f_1,t} \bigvee \neg W_{f_2,t}$$

It is vital to enforce action mutex constraints to ensure the correct semantics. Several algorithms are proposed to detect mutex between durative actions in temporal planning [118]. Here we compute those required action mutex for all transformed actions $a \in \mathcal{A}^s$ according to the method discussed in Chapter 2.

6.3 Temporally Expressive Planning, A SAS+ Perspective

Like classical planning, temporal planning can be formulated in a SAS+ style. A SAS+ temporal planning task is also consisted of multi-valued variables. A state in a SAS+ temporal planning task is thus an assignment to all multi-valued variables. We adopt the definition to transition (Definition 2). Based on transitions, Definition 23 defines duration action.

Definition 23 (SAS+ Durative Action). *A duration action o is defined by a 4-tuple $(\rho, o_{\vdash}, o_{\leftrightarrow}, o_{\dashv})$, where $\rho > 0$ is the duration of o, and others are sets of transitions. Suppose o is executed at time step t, then it ends at time step $t + \rho - 1$. The three sets of transitions are consequently defined as:*

- *$o_{\vdash}$ is a set of transitions that execute at time step t;*
- *$o_{\leftrightarrow}$ is a set of* prevailing *transitions that execute at all time step t', while $t < t' < t + \rho - 1$;*
- *$o_{\dashv}$ is a set of transitions that execute at time step $t + \rho - 1$.*

Note in previous section, $o_{\vdash}$ and $o_{\dashv}$ refer to simplified actions. Here under the context of SAS+ formulation, $o_{\vdash}$, $o_{\leftrightarrow}$ and $o_{\dashv}$ refer to sets of transitions. We use $M(o) = o_{\vdash} \cup o_{\leftrightarrow} \cup o_{\dashv}$ to indicate all the transitions enforced by o. Definition 23 enforces the equivalent semantics to that in Definition 19, although in different manners.

A durative action o is applicable iff: 1) Every transition in $o_{\vdash}$ is applicable to state s_t; 2) Every transition in $o_{\leftrightarrow}$ is applicable to state $s_{t'}$, for every t', such that $t < t' < t + \rho - 1$; 3) Every transition in $o_{\dashv}$ is applicable to state $s_{t+\rho-1}$. The effect of applying a durative action o is the effect of applying all the transitions in $M(o)$ to the corresponding states throughout the life time of o. In particular, because there are only prevailing transitions in $o_{\leftrightarrow}$, the effects are essentially enforced by $o_{\vdash}$ and $o_{\dashv}$. Base on Definition 23, a SAS+ temporal planning task is defined as follows.

Definition 24 (Temporal Planning Task). *A planning task Π in the SAS+ formalism is defined as a tuple $\Pi = \{\mathcal{X}, \mathcal{O}, s_{\mathcal{I}}, s_{\mathcal{G}}\}$, where*

- *$\mathcal{X} = \{x_1, \cdots, x_N\}$ is a set of* state variables*, each with an associated finite domain $Dom(x_i)$;*
- *$\mathcal{O}$ is a set of durative actions;*
- *A state s is a full assignment (a set of assignments that assigns a value to every state variable). If an assignment $(x = f)$ is in s, we can write $s(x) = f$. We denote $\mathcal{S}$ as the set of all states.*

- $s_{\mathcal{I}} \in \mathcal{S}$ *is the initial state, and* $s_{\mathcal{G}}$ *is a partial assignment of some state variables that defines the goal. A state* $s \in \mathcal{S}$ *is a goal state if* $s_{\mathcal{G}} \subseteq s$*.*

We use $\mathcal{T}$ to denote the set of all transitions in a planning task. We also use $R(x)$ to denote the set of all prevailing transitions related to x, and R the union of $R(x)$ for all $x \in \mathcal{X}$.

To handle concurrency, it is vital to forbid certain actions being executed in parallel, by enforcing mutual exclusions accordingly. First we adopt Definition 3 to define transition mutex. For simplicity, we say two transition sets T_1 and T_2 are mutual exclusive, as long as there exist two mutex transitions $\delta_1 \in T_1$ and $\delta_2 \in T_2$, such that δ_1 and δ_2 are mutex. Based on transition mutex, we have durative action mutex (also called durative mutex) defined in Definition 25.

Definition 25 (Durative Mutex). *Given two durative actions* o_1 *and* o_2*, they are mutual exclusive if there exist two transition* $\delta \in M(o_1)$ *and* $\delta' \in M(o_2)$*, such that* δ *and* δ' *are transition mutex.*

The definition to the durative actions's mutual exclusion is simple. Nevertheless, due to the complicated semantics of durative action, we do not want to try all the enumerations to find out mutex. Therefore, we check explicit conditions instead. Given two durative actions o and q, they are mutual exclusive if either of the following cases hold:

1. $o_{\vdash}$ and $q_{\dashv}$ are mutual exclusive, denoted as $o \bowtie_e^s q$.
2. $o_{\leftrightarrow}$ and $q_{\vdash}$ are mutual exclusive, denoted as $o \bowtie_s^o q$.
3. $o_{\leftrightarrow}$ and $q_{\dashv}$ are mutual exclusive, denoted as $o \bowtie_e^o q$.
4. $o_{\vdash}$ and $q_{\vdash}$ are mutual exclusive, denoted as $o \bowtie_s^s q$.
5. $o_{\dashv}$ and $q_{\dashv}$ are mutual exclusive, denoted as $o \bowtie_e^e q$.

Each individual of the five conditions above is by itself a sufficient condition for durative mutex. That is, if one condition holds for two given actions, then we know these two actions are mutual exclusive. Note that for the five relations, some are commutative (i.e. $\bowtie_s^s$ and $\bowtie_e^e$), while others are not. Given two durative action, they may have multiple mutex relations as above. In such cases, we need to explicitly encode all of them.

Given two intervals X and Y, there are seven possible relations between them [1]: X takes place before Y, X meets Y, X overlaps with Y, X starts Y, X during Y, X finishes Y, and X is equal

to Y. Since we are considering mutual exclusion, the relation 'takes place before' is impossible to hold. In addition, the relation "Overlaps" is covered by Conditions 2 and 3. Therefore, the five verification conditions covers all the possible conditions between two durative actions that are mutual exclusive. That is, if none of the five conditions holds for two durative actions, then they are not mutual exclusive.

6.4 A Transition Based Encoding for Temporal Planning

In this section, we introduce the encoding scheme for SAS+ temporal planning task Π. We have two types of variables:

1. Transition variables: $U_{\delta,t}, \forall \delta \in \mathcal{T}$ and $t \in [1, N]$, which may also be written as $U_{x,f,g,t}$ when δ is explicitly $\delta^x_{f \to g}$;
2. Action variables: $U_{o,t}, \forall o \in \mathcal{O}$ and $t \in [1, N]$.

As to constraints, we has eight classes of clauses for a task Π. In the following, we define each class for every time step $t \in [1, N]$ unless otherwise indicated.

(i). Initial state: $\forall x, s_{\mathcal{I}}(x) = f, \bigvee_{\forall \delta_{f \to g} \in \mathcal{T}(x)} U_{x,f,g,1}$;

(ii). Goal: $\forall x, s_{\mathcal{G}}(x) = g, \bigvee_{\forall \delta_{f \to g} \in \mathcal{T}(x)} U_{x,f,g,N}$;

(iii). Progression: $\forall \delta^x_{h \to f} \in \mathcal{T}$ and $t \in [1, N-1]$, $U_{x,h,f,t} \to \bigvee_{\forall \delta^x_{f \to g} \in \mathcal{T}(x)} U_{x,f,g,t+1}$;

(iv). Regression: $\forall \delta^x_{f \to g} \in \mathcal{T}$ and $t \in [2, N]$, $U_{x,f,g,t} \to \bigvee_{\forall \delta^x_{f' \to f} \in \mathcal{T}(x)} U_{x,f',f,t-1}$;

(v). Transition mutex: $\forall \delta_1 \forall \delta_2$ such that δ_1 and δ_2 are transition mutex, $\overline{U}_{\delta_1,t} \vee \overline{U}_{\delta_2,t}$;

(vi). Composition of actions $\forall o \in \mathcal{O}$:

$$U_{o,t} \to \bigwedge_{\forall \delta \in o_{\vdash}} U_{\delta,t}$$

$$U_{o,t} \to \bigwedge_{\forall \delta \in o_{\dashv}} U_{\delta,t+\rho-1}$$

$$U_{o,t} \to \bigwedge_{\forall \delta \in o_{\leftrightarrow}, t < t' < t+\rho-1} U_{\delta,t'};$$

(vii). Action existence: $\forall \delta \in \mathcal{T} \setminus R, U_{\delta,t} \rightarrow (\bigvee_{\forall o, \delta \in o_{\vdash}} U_{o,t} \vee \bigvee_{\forall o, \delta \in o_{\dashv}} U_{a,t-\rho+1});$

(viii). Action mutex: For each non-prevailing transition $\delta \in \mathcal{T}$ and time step t, let us define a set of variables $K_\delta = \{U_{o,t} \mid \delta \in o_{\vdash}\} \cup \{U_{o,t-\rho(o)+1} \mid \delta \in o_{\dashv}\}$. All the variables in K_δ are mutual exclusive to each other.

This encoding borrows ideas from SASE (Chapter 5) regarding the role of transitions in constructing constraints. Nevertheless there are essential differences, due to the semantics of temporal planning. The first major difference is how action mutex is enforced. The second difference is how actions and transitions match. In SASE a transition implies a disjunction of certain actions in the same time step. In the case of temporal planning, a transition may imply a durative action to be executed several time steps away.

Note that in Definition 25, we have five different (but not necessary disjoint) conditions to determine if two durative actions are mutual exclusive. While it comes to the phase of encoding, it becomes much easier because certain types of mutual exclusions are already handled by the clauses in Class v. For example, suppose we have two actions that are mutual exclusive on Condition 2 (i.e. $o \bowtie_s^o q$). In such a case, there exist transition $\delta \in o_{\leftrightarrow}$ and transition $\delta' \in q_{\vdash}$, such that δ and δ' are mutual exclusive. Such a mutex is covered by clauses of class v.

6.5 Experimental Results

Our experiments are done in a P2P domain that we develop and several other temporally expressive domains [24]. Besides the original problems, we also generate some larger instances from a problem generator that we develop. Note that we do not use all the domains in [24] because some of them cannot scale to large problems (e.g. the Match domain), and a few of them have variable-duration actions (e.g. the Café domain). The following is a brief description of the domains that are included in the experiments.

Note all the domains we test have required concurrency in them. High concurrencies in temporal planning problems are very different from most other temporal planning problems we have seen. Figure 6.1 illustrates the temporal dependencies (Definition 22) in several instances from different domains. All these instances have comparable problem sizes. The instance of the P2P domain has 90 facts and 252 actions, and the instance of Matchlift domain [24] has 216 facts and 558 actions. Figure 6.1 (I) is an instance of the Trucks domain, which is temporally simple and thus has all actions isolated. In Figure 6.1 (II) for the Matchlift domain, each action has up to two actions

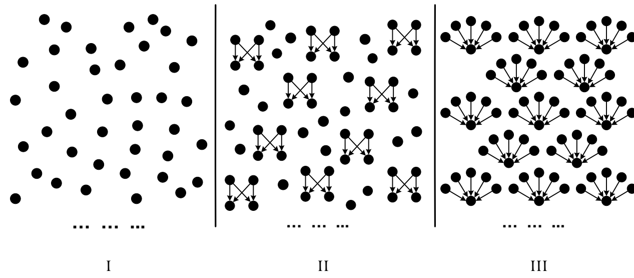


Figure 6.1: This figure partially illustrates temporal dependencies of actions for instances in three domains: Trucks, Matchlift and P2P. Each node represents an action. Each edge represents a temporal dependency between two actions.

temporally depending on it. In Figure 6.1 (III) for the P2P domain, each action has up to five actions temporally depending on it.

The experiments are presented in two parts. In the first part, we compare Crikey2, Crikey3, LPG-c [48], Temporal Fast Downward (TFD) [38], PET (planning as SAT with a STRIPS based encoding), and SET (planning as SAT with a SAS+ based encoding). Because PET and SET have the same solution quality, in Tables 6.1 to 6.4 we only list the solution quality for one.

6.5.1 The Peer-to-Peer Domain

This domain models file transfers in Peer-to-Peer (P2P) networks. In Peer-to-Peer (P2P) networks, each computer, called a peer, may upload or download data from another. One critical issue in P2P networks is that a substantial amount of inter-peer data communication traffic is unnecessarily duplicated. For those systems having consistent and intensive data sharing between peers, communication latency is a potential bottleneck of the overall network performance. Mechanisms in network design, particularly proxy caching (a.k.a. gateway caching), have been proposed to reduce duplicated data transmission. Making a good use of the proxy cache is critical for optimizing data transmission.

There are at least two different types of optimization in P2P networks. The first one is approached from the user's point of view: each individual user wants all the data needed within the shortest possible time [7]. The other type is approached from the point of view of a network service provider (such as an Internet service provider (ISP)), who owns the network but does not control individual peers. The main concern of a service provider is to reduce the overall communication load.

These two performance metrics are typically conflicting. We adopt performance metrics that lie in the middle of the above mentioned two. Under these metrics, the network owner knows each peer's needs, and the objective is to minimize the overall makespan for all the data delivery for all peers and minimize the total communication loads caused by different actions including serving and downloading. The problem, when casted as a planning problem, is temporally expressive.

The main constraint in this problem is to satisfy a file request from a peer p_1, the same file has to be offered by another peer p_2. p_1 can execute the *download* action to get the file, when 1) there is a route between p_1 and p_2, and 2) p_2 is *serving* the file throughout the transferring. As such, these *serve* and *download* actions require concurrency in any valid plan.

In addition, the proxy cache, which stores local caching files, will guarantee that, when p_2 is *serving* a file, any peer who is routed to p_2 can *download* the file very quickly. The upload bandwidth of a peer is typically much narrower than its download bandwidth. Therefore, enforced by the optimality goal, the more peers downloading this particular file, the larger the whole network's throughput will be, which brings about a shorter time span in a solution plan.

In the *serve* action, for example, the processing time of a file is proportional to its file size. We assume that by actively sharing a file, the uploading peer uses up its uploading bandwidth. That is, we assume that it cannot share another file simultaneously. This assumption will not impose a real restriction as we can introduce a time sharing scheme to extend the method we develop. A predicate 'serving' as one of the add-effects at the beginning indicates that the peer is sharing a file. When sharing a file from a peer, the connected route will guarantee that any other peers can get this file in a constant time (because download speed is much faster), as long as it is routed to the uploading peer.

The results on the P2P domain is shown in Table 6.1. The instances are generated randomly with different parameter settings, and the size of each file object is randomly chosen from four to eight units. The goal state for each instance is that each peer gets all requested files. There are two types of problems settings with different styles of network topology. One is loosely connected while the other is more highly connected.

Also, in the initial state, only leaf peers (those that are only connected to one other peer) have files to share. only connected to one other peer) have files to share. There are less concurrencies in this setting. Crikey3 is faster on two simpler instances but slower than PET and SET on two other larger instances. Overall, the time spans found by Crikey3 are about three to five times longer than those found by PET and SET. Crikey2 failed to solve any instance in this category. Instances 9 to 16 have more complicated network topology. Nearly all nodes are connected to one another. Every peer has some files needed by all others. In this setting, much higher concurrencies are required

P	Crikey3		PET			SET	
	Span	Time	Span	Time	Mem	Time	Mem
1	22	0.1	22	5.3	6	3.5	3
2	32	0.1	32	27.2	15	12.7	8
3	40	0.2	40	412.2	56	175.1	39
4	72	1.3	27	4.0	6	3.2	3
5	100	7.1	34	12.1	10	10.2	8
6	150	111.5	39	24.7	15	19.3	11
7	-		54	85.5	27	53.5	19
8	200	287.7	49	80.5	29	46.2	15
9	-		60	284.3	45	122.6	24
10	-		32	228.4	175	103.2	80
11	-		20	55.7	56	17.4	18
12	-		23	123.2	129	41.2	35
13	-		31	379.1	368	229.3	158
14	-		36	2377.6	793	711.4	266

Table 6.1: Results on the P2P domain. Crikey2, LPG-c and TFD fail to solve any of the instances.

to derive a plan. Both Crikey2 and Crikey3 fail to solve any instance within the resource limit. Crikey3 times out and Crikey2 reports no solution is found. It may be due to their incompleteness. SET consistently outperforms PET regarding both time and memory. In general SET uses half of the running time, and half of the memory usage. The advantage of using SET is especially clear on larger instances. For instance, it takes PET 2377.7 seconds to solve P2P-14, while it is just 711.0 for SET. To solve instance P2P-15, PET needs 633 MB memory, but SET only needs 138 MB.

6.5.2 The Matchlift Domain

In a Matchlift problem [24], an electrician enters an building to fix fuses during an outrage. Since there is no light, the electrician needs to light a match to make it possible to repair in a dark room. The required concurrencies between the action of lighting of a match, and the action of mending the fuse, make the problem temporally expressive. Furthermore, before mending a fuse, the electrician may need to travel through the building by taking the elevator to the appropriate floor, and then find out and enter the correct room.

The original Matchlift domain [24] has some flaws, in which an electrician's position is not updated until the end of a durative action. This can introduce a huge increase to the number of electricians needed in Crikey2 and Crikey3, and eventually electricians will exist everywhere. To make Crikey2 and Crikey3 work properly in this domain, we fix the flaws and use the fixed version of Matchlift domain for this set of experiments.

P	Crikey2		Crikey3		LPG-c		TFD		PET			SET	
	Span	Time	Span	Time	Span	Time	Span	Time	Span	Time	Mem	Time	Mem
1	13	3.0	18	0.1	17	0.1	13	0.0	13	2.2	4	1.6	5
2	11	0.7	14	0.3	9	12.6	9	0.1	9	1.5	1	1.2	5
3	23	2.9	28	0.1	-		22	0.0	23	8.8	7	7.2	6
4	19	9.6	34	0.1	-		18	0.0	21	8.5	10	7.8	10
5	35	52.8	43	0.1	-		24	0.0	28	21.0	17	24.9	18
6	39	24.4	47	1.4	-		25	8.3	29	54.5	82	101.5	41
7	37	131.9	58	0.4	-		31	1.2	37	3042.2	361	507.2	117
8	42	73.8	58	1.9	-		30	20.8	34	1027.3	325	945.9	151
9	39	60.4	43	0.1	-		26	7.3	30	46.7	58	60.1	51
10	28	234.1	58	0.1	-		28	0.0	28	244.1	235	204.9	103
11	47	376.7	58	0.7	-		31	0.0	37	3483.5	659	977.9	241

Table 6.2: Results on the Matchlift domain.

The results on the Matchlift domain are in Table 6.2. We generate all instances randomly using different parameters for the numbers of floors, rooms, matches and fuses. Each instance has the same number of fuses and matches. In other words, these instances are easier because we can always find a valid plan, such that there is exactly one running action concurrent with a lighting match action. On all instances, Crikey3 is the fastest to find solutions, but with the poorest quality. TFD is slightly slower but the plan quality is better. LPG-c cannot handle the concurrency that it only solves 2 out of the 11 instances. PET spends more running time, since it finds the optimal solutions. SET further has better efficiency than PET do. The running time is up to six times faster and the memory consumption is as low as 1/3.

6.5.3 The Matchlift-Variant Domain

The original Matchlift domain only requires one electrician to do the repairing. Also, there are always enough matches available. It is of a relatively weak form of required concurrency. We make a revised Matchlift domain (called Matchlift-Variant domain), which requires more concurrencies due to two changes. First, the number of matches is less than the number of fuses, so that multiple electricians need to share one match. Second, we reduce the duration of the '*mend_fuse*' action so that an electrician is able to conduct more mending actions during one match's lighting, which also results in higher concurrencies.

The results on the Matchlift-Variant domain are shown in Table 6.3. All instances are generated with increasing numbers of fuses and electricians. All other settings are random. Instances with the same number of fuses and electricians might still have different degrees of concurrency, due to different numbers of matches and other resources available. For example, although Instances 7 and 8 have the same parameters, Instance 8 is more difficult than Instance 7 due to the different ways how the fuses are distributed over the rooms.

P	Crikey2		Crikey3		LPG-c		TFD		PET			SET	
	Span	Time	Span	Time	Span	Time	Span	Time	Span	Time	Mem	Time	Mem
1	14	10.9	17	5.1	-		-		13	3.5	3	3.4	5
2	13	147.7	16	7.5	13	0.4	-		13	1.7	3	2.1	4
3	19	6.1	23	0.1	-		19	0.5	18	6.9	7	5.9	6
4	25	106.0	27	41.8	21	460.9	-		21	15.7	16	15.7	11
5	23	20.3	33	0.1	-		29	5.9	22	19.8	21	16.2	10
6	25	121.6	27	42.0	33	0.1	-		21	16.9	18	15.0	11
7	-		TLE		-		-		16	46.2	21	14.3	10
8	17	167.1	TLE		-		-		16	168.1	29	110.0	23
9	TLE		TLE		-		33	104.7	22	70.4	81	34.1	21
10	TLE		TLE		-		-		20	239.2	39	157.3	35
11	TLE		TLE		-		-		16	2035.0	85	135.9	38
12	TLE		TLE		-		-		13	3.7	6	3.2	7

Table 6.3: Results on the Matchlift-Variant (MLR) domain.

As shown by the experimental results, PET and SET find optimal solutions on all instances tested, whereas Crikey2 and Crikey3 ran out of time on most instances and generate suboptimal plans on the few instances they finish. For the instances they solve, Crikey3 has the worst solution quality. It is very efficient in finding a solution in a few small instances, but in other instances, it is even slower. LPG-c and TFD can only handle very few instances in this domain.

The results on Instances 11 and 12 are special and interesting. These two instances are generated under the same parameter setting, except for the number of matches. Instance 12 has only one match, which means the four electricians need to cooperate with each other perfectly to get all the fuses fixed. Comparing to Instance 11, Instance 12 turns out to be more difficult for Crikey2, because it requires more concurrencies. Instance 12 is much easier than Instance 11 for PET and SET. PET solves Instance 12 in just about three seconds, but spent more than 1800 seconds on Instance 11. The difference is less significant for SET, which solves Instance 12 in 3 seconds, and Instance 11 in 135 seconds.

6.5.4 The Driverslogshift Domain

The Driverslogshift domain [24] is an extended version of the Driverslog domain from IPC-3 [127]. It does similar things as those defined in the original Driverslog domain, as long as the worker is in the '*working*' status. The *working* status, for each individual worker, is modeled as a durative action with a fixed duration. After the *working* action is over, the worker needs to take a rest, which takes a constant duration. The *working* action has to be concurrent with other actions by the worker. This is why the problem is temporally expressive. The possible actions of a worker, are driving trucks between locations, loading/unloading the trucks, and walking between locations.

P	Crikey2		Crikey3		LPG-c		PET			SET	
	Span	Time	Span	Time	Span	Time	Span	Time	Mem	Time	Mem
1	122	16.7	224	0.1	712	336.7	102	134.9	37	104.8	23
2	122	4.0	122	0.1	244	2.2	122	177.5	31	129.2	30
3	122	18.8	225	0.1	346	712.5	122	198.5	38	140.0	31
4	122	19.8	323	0.2	122	365.3	122	200.9	38	137.2	31
5	102	38.3	238	0.1	224	653.0	102	183.7	48	131.7	35
6	122	10.4	326	0.1	224	85.8	118	186.7	39	123.5	24
7	102	201.4	102	0.2	102	9.3	102	319.0	79	209.3	60
8	102	180.4	125	0.2	102	2.2	102	322.1	75	228.1	80
9	102	159.5	125	0.2	102	15.9	102	318.1	79	238.0	97

Table 6.4: Results on the Driverslogshift domain. TFD cannot solve any instance of this domain.

The problem instances in the Driverslogshift domain have much longer makespan than those in the Matchlift and P2P domains.

Compared with P2P and Matchlift, this domain has long durative actions, which give rise to a long makespan. Therefore, it is relatively difficult to optimally solve instances in this domain.

The problem instances in the Driverlogshift domain have much longer time spans than those in the Matchlift and P2P domains. The actions with duration of two are changed to three to distinguish $\vdash$, $\dashv$ and $\leftrightarrow$ conditions and effects. This change is made to accommodate PET and SET for solving discrete problems. This domain is different from P2P and Matchlift. It has long durative actions, which give rise to longer time spans. Therefore, it is relatively difficult to optimally solve instances in this domain. These observations are reflected by our experimental result in Table 6.4.

As shown in Table 6.4, the optimal time spans of the instances tested, provided by PET and SET, are typically much shorter than those by Crikey3. For example, the optimal time span for Instance 4 is about one third of the time span reported by Crikey3. As a trade-off, both SET and PET need longer time for finding optimal solutions. Now consider Crikey2, a suboptimal solver. Surprisingly, it is able to generate solutions of the same quality as what STEP found on most instances in this domain. SET is in general better than PET, but the improvements are not as significant as those in previous domains. Although LPG-c can solve all the instances, it however is the worst in both planning quality and running time.

6.5.5 Encoding Efficiency

In addition to the number of instances solved, we compare the two encoding schemes regarding their efficiency. In Figure 6.2, we present the number of instances that PET and SET can solve, with increasing limits on number of variables or clauses. In both cases, SET in general solves 50%

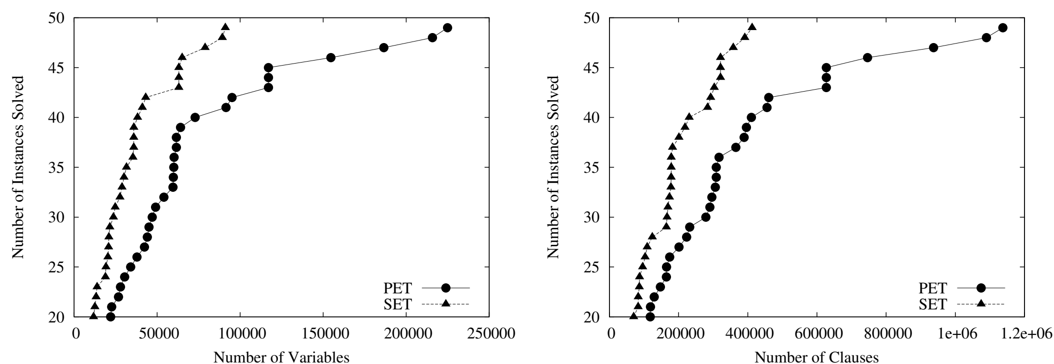


Figure 6.2: The Number of instances that PET and SET can solve, with increasing limits on number of variables or clauses.

more instances than PET. PET uses 250,000 variables to solve all the problems and it only costs SET 100,000 variables. For the number of clauses, it takes PET 1,200,000 clauses to solve all the instances and only 400,000 for SET. Since SET is more compact by having fewer variables and fewer clauses, it consistently uses less memory for all the instances.

6.6 Summary

We have presented a planning as SAT methodology for temporal planning. Two encodings are introduced, one is based on STRIPS and the other on SAS+. Comparing to the state-of-the-art search based planners, PET and SET can handle the required concurrency much better. They both solve instances that state-of-the-art planners cannot handle at all. For these two encodings, the experimental results on the temporally expressive domains show that the encoding based on SAS+ is more time and memory efficient.

Chapter 7

Cost Sensitive Temporally Expressive Planning

In this chapter, we extend our temporal planning approach to further handle one more advanced feature: action cost. Chapter 6 shows that most existing temporal planners are temporally simple without required concurrency, which is crucial for many applications. Although there are a few planners that can handle required concurrency and some planners can handle action costs, none of them can handle both. Existing temporal planners either attempt to minimize the total duration of the solution plan (i.e. makespan), or do not consider any quality metric at all. Nevertheless, many applications want to optimize not only the makespan, but also the total action cost [31], which can be used to represent features, such as cost of resources used, the total money spent, and the total energy consumed. Action cost is adopted as a new criterion in IPC-6 planning competition [122].

We call such problems with both temporal expressiveness and action cost, as Cost-Sensitive Temporally Expressive (CSTE) planning. CSTE planning is important and ubiquitous in many real world scenarios. Example CSTE domains include but not limit to:

1. **Web service composition.** Web service composition (WSC) is the problem of integrating multiple web services to satisfy a particular request [98]. Planning has been adopted as one of the major methods for WSC [18, 98]. WSC problems may require CSTE planning, since different web services operate under different conditions and different rates of cost (some are free). As a result, it is desirable to optimize the QoS metrics, such as total price, reliability, and reputation. Moreover, temporally concurrent actions are often needed to coordinate multiple web services.

2. **Peer-to-Peer network communication.** In Peer-to-Peer network communication, one peer's uploading has to be concurrent with one or more other peers' downloading [64]. Besides the required concurrency, modern communication is service oriented; communication actions

are charged by different costs, depending on the types of network service used. A desirable planner will need to find temporally expressive solutions that also minimize the total action costs and thus require a CSTE planning.

3. **Autonomous systems.** Planning for autonomous systems, including robotics, rovers, and spacecrafts, often requires CSTE planning. Consider a spacecraft controlling example [117] in which the spacecraft movement is made by firing thrusters. Multiple operations need to be performed within the time interval when the thrusters are fired, thus requiring action concurrency. Moreover, operation costs such as energy need to be minimized in order to best utilize the on-board resources.

We extend our planning approach in Chapter 6 to handle CSTE planning tasks. Central to this approach is a transformation for turning a CSTE instance into an optimization problem with SAT constraints. Such a problem, called a MinCost SAT formulation, is a SAT problem with an objective of minimizing the total cost of literals assigned to be true [83].

Given a MinCost SAT instance compiled from a CSTE planning task, we develop two approaches to solve it. In the first approach, we compile a MinCost SAT into Weighted Partial Max-SAT problems and apply existing Max-SAT solvers. Second, we develop BB-DPLL, a branch-and-bound algorithm based on DPLL, to directly solve MinCost SAT problems. To this end, an effective bounding technique, and an action-cost-based variable branching scheme is developed to make the problem solving efficient. Our results show that such a SAT-based approach is a good choice for CSTE planning.

The rest of this chapter is organized as follows. In Section 7.1 we define CSTE planning. We discuss the method of using Max-SAT to solve CSTE planning tasks in Section 7.2, and the branch-and-bound planning specialized algorithm in Section 7.3. Finally, we present our experimental results on a variety of CSTE planning domains in Section 7.4.

7.1 Cost Sensitive Temporal Planning Task

A durative action with cost is defined by a tuple $(\rho, \mu, \pi_{\vdash}, \pi_{\leftrightarrow}, \pi_{\dashv}, \alpha_{\vdash}, \alpha_{\dashv})$, such that μ is the action cost and all others follow Definition 19. A cost sensitive temporally expressive planning task is defined by $\Psi = (\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$, which is similar to the definition to temporal planning tasks in Section 6.1, except $\mathcal{A}$ is a set of durative actions with costs.

We adopt the overall procedure for temporal planning (Section 6.2). In the first step, each cost-sensitive durative action o is converted into two simple actions and one propositional fact, written as $(o_{\vdash}, o_{\dashv}, f^o)$. We use the symbol a to denote the simple action which indicates the starting $(a = o_{\vdash})$ or ending events $(a = o_{\dashv})$ of o. The fact f^o, when is true, indicates that o is being executed. We denote the set of all such f^o as $F^o = \{f^o \mid o \in \mathcal{A}\}$. Similarly, this transformation would only take effects on those actions with $\rho > 1$.

Given a CSTE planning task $\Psi = (\mathcal{F}, \mathcal{A}, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$, we denote the simplified task as $\Psi^s = (\mathcal{F}^s, \mathcal{A}^s, \varphi_{\mathcal{I}}, \varphi_{\mathcal{G}})$, where $F^s = F \cup F^o$ and $O^s = \{o_{\vdash}, o_{\dashv} \mid o \in \mathcal{A}\} \cup \{$no-op $action\ for\ f \mid f \in F^s\}$. The cost of each simple action is defined by

$$\mu(a) = \begin{cases} \mu(o), & \text{if } a = o_{\vdash} \\ 0, & \text{otherwise} \end{cases}$$

Once we have the temporal planning task compiled, we can apply the encoding introduced in Section 6.2. For a CSTE planning task, the resulting SAT instance is a MinCost SAT problem, defined as follows.

Definition 26 (MinCost SAT Problem). *A MinCost SAT problem is a tuple $\Phi^c = (V, C, \mu)$, where V is a set of Boolean variables, C is a set of clauses, and μ is a function $\mu : V \to \mathbb{N}$. A solution to Φ is an assignment Γ that minimizes the objective function:*

$$cost(\Gamma) = \sum_{x \in V} \mu(x)\Gamma(x),$$

subject to: $\quad \Gamma(p) = 1, \forall p \in C.$

Our encodings introduced in Chapter 6 can be easily extended to model action costs, by assigning the variable of a starting action $o_{\vdash}$ with a cost of $\rho(o)$. Such a flexibility is one of the advantages of planning as SAT approach. To solve the MinCost SAT problems, we apply two approaches as discussed in the following sections.

7.2 Solve CSTE by Weighted Partial Max-SAT

Besides MinCost SAT, there is another type of extended SAT problem, called weighted partial Max-SAT Problem [125], which is widely accepted by the SAT community. There are a lot of well

developed Max-SAT solvers. We compile a MinCost SAT problem into a Max-SAT problem, and make use of existing Max-SAT solvers.

Definition 27 (Weighted partial Max-SAT Problem). A weighted partial Max-SAT problem is a tuple $\Phi^a = (V, C^h, C^s, w)$, where V is a set of variables, C^h and C^s are sets of hard and soft clauses, respectively, and w is the weight function of soft clauses defined by $w : C^s \rightarrow \mathbb{N}$.

A solution to Φ^a is a variable assignment Γ that maximizes the function:

$$weight(\Gamma) = \sum_{p \in C^s} w(p)\Gamma(p),$$

$$\text{subject to:} \qquad \Gamma(p') = 1, \forall p' \in C^h.$$

A weighted partial Max-SAT problem Φ^a amounts to finding a variable assignment, such that all hard clauses are satisfied, and the total weight of satisfied soft clauses is maximized. In the following, we use the notation of assignment function $\Gamma() : V \cup C \rightarrow \{0, 1\}$ in the following definitions.

Given a MinCost SAT instance $\Phi^c = (V, C, \mu)$, we construct a weighted partial Max-SAT instance $\Phi^a = (V, C, C^s, w)$. The hard clause set is equivalent to the clause set in the original MinCost SAT problem. The soft clause set C^s is constructed as: $C^s = \{\neg x \mid x \in V\}$. For each clause $p \in C^s$, its weight is consequently defined as: $w(p) = \mu(x)$, where $p = \neg x$.

According to Definition 27, given a variable assignment Γ, the objective function of the weighted partial Max-SAT instance is:

$$\begin{aligned}
weight(\Gamma) &= \sum_{\forall p \in C^s} w(p)\Gamma(p) \\
&= \sum_{\forall x \in V} \mu(x)\Gamma(\neg x) \ (\because C^s = \{\neg x \mid x \in V\}) \\
&= \sum_{\forall x \in V} \mu(x)(1 - \Gamma(x)) \\
&= \sum_{\forall x \in V} \mu(x) - \sum_{\forall x \in V} \mu(x)\Gamma(x) \\
&= \sum_{\forall x \in V} \mu(x) - cost(\Gamma).
\end{aligned}$$

Hence, maximizing $weight(\Gamma)$ is equivalent to minimizing $cost(\Gamma)$. Since $cost(\Gamma)$ is the objective function of the MinCost SAT problem Φ^c, solving a MinCost SAT problem is equivalent to solving the corresponding Max-SAT problem Φ^a. To take an example, let us consider a MinCost SAT

problem $\Phi^c = (V, C, \mu)$:

$$C\ : x_1 \ \vee \ x_2; \ \ x_2 \ \vee \ \neg x_3$$

$$c\ : \mu(x_1) = 5; \ \ \mu(x_2) = 10; \ \ \mu(x_3) = 7$$

The corresponding weighted partial Max-SAT problem $\Phi^a = (V, C^h, C^s, w)$ will be:

	$Clause$	$w(p)$
C^h :	$x_1 \ \vee \ x_2$	∞
	$x_2 \ \vee \ \neg x_3$	∞
C^s :	$\neg x_1$	5
	$\neg x_2$	10
	$\neg x_3$	7

The optimal solution to Φ^c is $\Gamma(x_1) = 1$, $\Gamma(x_2) = 0$, $\Gamma(x_3) = 0$, and the objective function $cost(\Gamma) = 5$. The optimal solution to Φ^a is the same Γ, and $weight(\Gamma) = 17$. By solving any of the two problems, we have the solution to the other. Since Max-SAT has been extensively studied [132, 43, 110, 113, 28, 81, 96], we can make use of the existing Max-SAT solvers to solve the MinCost SAT instances.

7.3 A Branch-and-Bound Algorithm

In this section, we develop a specialized branch-and-bound based DPLL (BB-DPLL) algorithm for a MinCost problem encoded from a CSTE planning task. Based on the standard branch-and-bound procedure, we introduce two key planning specific techniques: a cost bounding mechanism based on relaxations (Section 7.3.1) and a variable branching scheme based on action costs (Section 7.3.2). These two techniques together improve the problem solving efficiency.

Here we give an overview of the BB-DPLL procedure (Algorithm 9), which integrates branch and bound search strategy into the DPLL procedure. It uses a propagation queue that contains all literals that are pending for propagation and also contains a representation of the current assignment.

Algorithm 9: BB-DPLL(Φ^c)

Input: MinCost SAT problem Φ^c
Output: a solution with minimum $cost$

```
1  cost_init() ;
2  τ ← ∞, num ← 0;
3  while true do
4      conflict ← propagate();
5      if conflict then
6          learnt ← analyze(conflict);
7          if conflict is of top-level then return num > 0 ? SAT:UNSAT;
8          add learnt to the clause database and backtrack();
9      else
10         cost_propagate() ;
11         g(Γ) ←cost(Γ) ;
12         if g(Γ) + h(Γ) ≥ τ then backtrack();
13         if all variables are assigned then
14             num++ ;
15             τ ← cost(Γ) ;
16             backtrack();
17         else
18             decide();
```

BB-DPLL repeatedly propagates the literals in the propagation queue and returns a conflict if there is any (Line 4). Once a conflict occurs, the procedure analyze() checks the conflict to generate a learned clause (Line 6); after that, it calls backtrack() to undo the assignment until exactly one of the literals in the learned clause becomes unassigned (Line 8). If no conflict occurs, it calls the cost_propagate() procedure to estimate the lower bound of current assignment (Line 10). It prunes a search node if the lower bound of its cost exceeds τ, the cost of the incumbent (currently best) solution (Line 12), or calls decide() to select a unassigned variable, assigns it to be $true$ or $false$, and inserts it into the propagation queue (Line 18).

Each time a satisfying solution is found when there is no unassigned variable any more, BB-DPLL updates the incumbent solution, including solution number num and threshold τ, and then backtracks (Line 14-16). BB-DPLL keeps searching the whole space until all satisfiable solutions are either visited or pruned, in order to find the one that minimizes $cost()$, the objective function of the MinCost SAT problem. The procedure stops when a top level conflict is found.

7.3.1 Lower Bounding Based on Relaxed Planning

The lower bounding function is a key component in a branch-and-bound algorithm. Given a partial variable assignment Γ, we can compute a lower bound of the costs of any solutions based on this

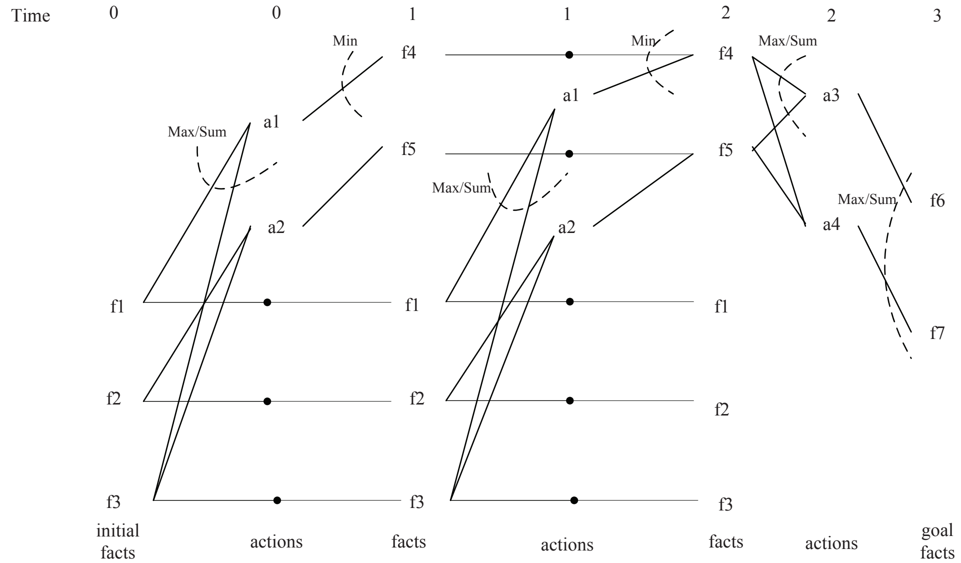


Figure 7.1: A relaxed planning graph for a simple example with 4 actions and 7 facts. For simplicity, no-ops are represented by dots and some action nodes in time steps 1 and 2 are ignored. $\mu(a_1, a_2, a_3, a_4) = (10, 10, 15, 5)$.

partial assignment. A typical lower bounding function is $f(\Gamma) = g(\Gamma) + h(\Gamma)$, where $g(\Gamma)$ is the total action costs of those variables already assigned to be true in Γ, and $h(\Gamma)$ is a lower bound on the pending cost that will incur by those variables unassigned in Γ. In a basic BB-DPLL algorithm, we simply set $h(\Gamma)$ to zero. In such a setting, the lower bound is exactly $g(\Gamma)$.

However, the lower bound by this basic scheme produces is too loose. Our lower bounding function is based on the idea of integrating max-heuristic rule with additive/sum-heuristic rule when facts are additive/independent. These heuristics have been extensively studied in state space search planners (such as HSP-r [9] and AltAlt [93]) and SAT solvers (such as MinCostChaff [44] and DPLL$_{BB}$ [80]). The implementation of the max-sum bounding function is customized for CSTE planning based on the relaxed planning graph [10, 62].

We first construct a relaxed planning graph [92] and compute the lower bound cost $h(x)$ of each variable x in the graph. Then, we compute $h(\Gamma)$ for a partial assignment based on $h(x)$. We also prove that this lower bounding function is admissible. That is, $h(\Gamma)$ is always a lower bound of the pending cost of any Γ. Figure 7.1 shows a running example of relaxed planning graph.

Let us define two sets, contribution set and additive set, which are useful for defining an accurate bounding function. Then, we define a lower bounding function $h(x)$ for each variable x and a function $h(\Gamma)$ for any partial assignment Γ. After that, we prove that $h(\Gamma)$ is always a lower bound of the pending cost of any Γ.

For each variable $x \in V$, the contribution set $cont(x)$ (formally defined below) is the set of all possible actions in any solution plan that reaches the assignment $v_\Gamma(x) = 1$ from the initial state I.

Definition 28 (Contribution Set). Given a problem $\Pi^s = (\mathcal{F}^s, \mathcal{A}^s, \varphi_\mathcal{I}, \varphi_\mathcal{G})$ transformed from a CSTE problem $\Pi = (\mathcal{F}, \mathcal{A}, \varphi_\mathcal{I}, \varphi_\mathcal{G})$, and the corresponding MinCost SAT problem $\Phi^c = (V, C, \mu)$ with makespan N, the contribution set **cont**(x) of variable x is defined as:

- if $x = x_{f,t} \in V (0 \le t \le N)$:

$$cont(x_{f,t}) = \begin{cases} \bigcup\limits_{\{a \mid f \in add(a)\}} cont(x_{a,t-1}), & t > 0 \\ \emptyset, & t = 0 \end{cases}$$

- if $x = x_{a,t} \in V (0 \le t < N)$:

$$cont(x_{a,t}) = \begin{cases} \bigcup\limits_{f \in pre(a)} cont(x_{f,t}), & \text{if a is a no-op action} \\ \bigcup\limits_{f \in pre(a)} cont(x_{f,t}) \cup \{a\}, & \text{otherwise} \end{cases}$$

The above definition gives rise to an effective algorithm for computing $cont(x)$ for every variable $x \in V$ in a preprocessing phase.

Definition 29 (Additive Set). A variable set $X = \{x_{1,t_1}, x_{2,t_2}, ..., x_{n,t_n}\}$ is an additive set, denoted as $adt(X)$, if for each variable pair (x_{i,t_i}, x_{j,t_j}), $x_{i,t_i}, x_{j,t_j} \in X$ and $i \neq j$, we have $cont(x_{i,t_i}) \cap cont(x_{j,t_j}) = \emptyset$.

Since the contribution set of a variable x contains all actions in any possible plan that reaches x from the initial state, for an additive set $X = \{x_{1,t_1}, x_{2,t_2}, ..., x_{n,t_n}\}$, there is no common action in any two plans reaching x_{i,t_i} and x_{j,t_j}, respectively. In other words, given an additive set X and two variables $x_{i,t_i}, x_{j,t_j} \in X (i \neq j)$, if there exists an action a as a common action in two plans reaching x_{i,t_i} and x_{j,t_j}, then $a \in cont(x_{i,t_i}) \cap cont(x_{j,t_j})$, which contradicts Definition 29. For example, in Figure 7.1, we $cont(x_{f_4,2}) \cap cont(x_{f_5,2}) = \{a_1\} \cap \{a_2\} = \emptyset$, thus $\{x_{f_4,2}, x_{f_5,2}\}$ is an additive set.

At each decision point (corresponding to a partial assignment Γ) during the search, for each variable x, $h(x)$ is a lower bound of the following quantity: the minimum total action costs of any solution plan that: 1) reaches the assignment $\Gamma(x) = 1$ from the initial state $\varphi_\mathcal{I}$, and 2) is consistent with the partial assignment Γ.

Definition 30 Given a partial assignment Γ, for each variable $x \in V$, if $\Gamma(x) = 0$, we let lower bounding function $h(x)$ to be ∞. If $\Gamma(x) = 1$ or x is unassigned, we check x is either a fact variable or an action variable, and define $h(x)$ accordingly as follows:

$$h(x_{f,t}) = \begin{cases} \min\limits_{\{a|f \in add(a)\}} h(x_{a,t-1}), & t > 0 \\ 0, & t = 0 \text{ and } f \in \varphi_{\mathcal{I}} \\ \infty, & t = 0 \text{ and } f \notin \varphi_{\mathcal{I}} \end{cases}$$

$$h(x_{a,t}) = \mu(x_{a,t})\alpha(x_{a,t}) + \begin{cases} \sum\limits_{f \in pre(a)} h(x_{f,t}), & \text{if } adt(\{x_{f,t}|f \in pre(a)\}) \\ \max\limits_{f \in pre(a)} h(x_{f,t}), & \text{otherwise} \end{cases}$$

where $\alpha(x_{a,t}) = 0$ if $\Gamma(x_{a,t}) = 1$, otherwise $\alpha(x_{a,t}) = 1$.

For a variable x assigned to be false, since no solution plan satisfying Γ can reach $v_\Gamma(x) = 1$, we have $h(x) = \infty$. The lower bound of a non-false assignment fact variable $x_{f,t}$ is the minimum estimated value of action variables $\{x_{a,t-1}|f \in add(a)\}$. The necessary condition for $x_{a,t}$ to be true is that all of a's precondition variables are true. Thus, a lower bound for $h(x_{a,t})$ is the maximum of the h values of a's precondition variables. Further, if a's precondition set is *additive*, the lower bound can be improved by summing up the h values of a's precondition variables. $\alpha(x)$ makes the variable cost $\mu(x)$ only be counted once in $h(x)$ or $g(\Gamma)$.

Based on the h values for variables, we now define the **lower bounding function** $h(\Gamma)$ for any partial assignment Γ. $h(\Gamma)$ is computed as

$$h(\Gamma) = \begin{cases} \sum\limits_{f \in G} h(x_{f,N}), & \text{if } adt(\{x_{f,N}|f \in \varphi_{\mathcal{G}}\}) \\ \max\limits_{f \in G} h(x_{f,N}), & \text{otherwise} \end{cases} \tag{7.1}$$

For example, in Figure 7.1, since *adt(*$\{x_{f_4,2}, x_{f_5,2}\}$ *)*, $h(x_{a_3,2}) = \mu(x_{a_3,2})\alpha(x_{a_3,2}) + h(x_{f_4,2}) + h(x_{f_5,2})$ and $h(x_{a_4,2}) = \mu(x_{a_4,2})\alpha(x_{a_4,2}) + h(x_{f_4,2}) + h(x_{f_5,2})$. At the beginning of search, $\alpha(x_{a,t}) = 1$ for all $x_{a,t} \in V$. Then, $h(x_{a_3,2}) = 35$ and $h(x_{a_2,2}) = 25$. $h(\Gamma)$ is $max\{h(x_{f_6,3})$, which is $h(x_{f_7,3})\} = max\{h(x_{a_3,2}), h(x_{a_2,2})\} = 35$. Note that we would get a worse lower bound without considering additive set, in which case we will get $h(x_{a_3,2}) = 25$, $h(x_{a_2,2}) = 15$ and $h(\Gamma) = 25$.

Let us prove that $h(\Gamma)$ is a lower bound of the actual minimum pending cost. Given a partial assignment Γ and an arbitrary variable $x \in V$, we use $\mu^p(x)$ to represent the **pending cost** (the total

cost of all action variables that are unassigned in Γ) of any solution plan that assigns $\Gamma(x) = 1$, and otherwise exactly the same to Γ. $\mu^p(x)$ is unbounded if $\Gamma(x) = 1$ cannot be reached in any solution within the predefined makespan bound consistent with Γ, denoted as $\mu^p(x) = \infty$; otherwise, $\mu^p(x)$ is bounded.

Lemma 5 *Given a partial assignment Γ, for each fact variable $x_{f,t} \in V(0 \leq t \leq N)$ such that $\mu^p(x_{f,t})$ is bounded, we have $\mu^p(x_{f,t}) \geq h(x_{f,t})$.*

Proof We prove this lemma by induction.

Basis. When $t = 0$, for each fact variable $x_{f,0}$, we have

$$\mu^p(x_{f,0}) = \begin{cases} 0, & f \in \varphi_I \\ \infty, & f \notin \varphi_I \end{cases}$$

Thus, both $\mu^p(x_{f,0})$ and $h(x_{f,0})$ are 0. The claim is satisfied.

Induction. Suppose that $\mu^p(x_{f,t}) \geq h(x_{f,t})$ for all bounded $\mu^p(x_{f,t})$ when $t \leq k$, we will prove $c^p(x_{f,k+1}) \geq h(x_{f,k+1})$, for any $x_{f,k+1} \in V$ with bounded $\mu^p(x_{f,k+1})$.

For each action variable $x_{o,k}$ with bounded $\mu^p(x_{o,k})$, to make $x_{o,k}$ true, all fact variables $\{x_{f,k} | f \in pre(o)\}$ have to be true and each $\mu^p(x_{f,k})$ must also be bounded. We thus have

$$\begin{aligned} \mu^p(x_{o,k}) &\geq \mu(x_{o,k})\alpha(x_{o,k}) + \max_{f \in pre(o)} \mu^p(x_{f,k}) \\ &\geq \mu(x_{o,k})\alpha(x_{o,k}) + \max_{f \in pre(o)} h(x_{f,k}) \end{aligned}$$

We use maximum instead of summation over all precondition variables since there may be some common action variables counted more than once in different $\mu^p(x_{f,k})$ for different precondition variables $x_{f,k}$. But in the special case when $\{x_{f,k} | f \in pre(o)\}$ is an additive set, we have

$$\begin{aligned} \mu^p(x_{o,k}) &= \mu(x_{o,k})\alpha(x_{o,k}) + \sum_{f \in pre(o)} \mu^p(x_{f,k}) \\ &\geq \mu(x_{o,k})\alpha(x_{o,k}) + \sum_{f \in pre(o)} h(x_{f,k}) \end{aligned}$$

According to Definition 30, we have $\mu^p(x_{o,k}) \geq h(x_{o,k})$ for any $o \in O^s$ with bounded $\mu^p(x_{o,k})$.

For each fact variable $x_{f,k+1}$ with bounded $\mu^p(x_{f,k+1})$, to make $x_{f,k+1}$ true, at least one action variable $x_{o,k|f \in add(o)}$ has to be true. We consequently have:

$$\begin{aligned}\mu^p(x_{f,k+1}) &\geq \min_{\{o|f \in add(o)\}} \mu^p(x_{o,k}) \\ &\geq \min_{\{o|f \in add(o)\}} h(x_{o,k}) \\ &\geq h(x_{f,k+1}).\end{aligned}$$

This lemma is thus proved. □

For any solution plan p reaching all goal variables $\{x_{f,N}|f \in \varphi_{\mathcal{G}}\}$ from the initial state $\varphi_{\mathcal{I}}$ and satisfying the current partial assignment Γ, we denote the **pending cost** of the plan as $\mu^p(\Gamma)$, which is the total cost of all the actions that are not assigned in Γ. The **minimum pending cost** for any partial assignment Γ, denoted as $h^r(\Gamma)$, is the minimum $\mu^p(\Gamma)$ over all possible solution plans that are consistent with Γ, i.e. $h^r(\Gamma) = \min_p \mu^p(\Gamma)$.

Theorem 4 *Given a planning task $\Pi^s = (\mathcal{F}^s, \mathcal{A}^s, \varphi_I, \varphi_G)$ transformed from a CSTE planning task $\Pi = (\mathcal{F}, \mathcal{O}, \varphi_I, \varphi_G)$, and the corresponding MinCost SAT problem $\Phi^c = (V, C, \mu)$, for any partial assignment Γ of Φ^c, we have $h(\Gamma) \leq h^r(\Gamma)$.*

Proof If there is no solution plan satisfying Γ, we have $h^r(\Gamma) = \infty \geq h(\Gamma)$. Otherwise, $h^r(\Gamma)$ is bounded and there exists some bounded $\mu^p(\Gamma)$. Consider any solution plan p reaching all goals and satisfying Γ, we have

$$\begin{aligned}\mu^p(\Gamma) &\geq \max_{f \in \varphi_{\mathcal{G}}} \mu^p(x_{f,N}) \\ &\geq \max_{f \in \varphi_{\mathcal{G}}} h(x_{f,N}) \ (Lemma\ 5)\end{aligned}$$

Furthermore, if the goal variables $\{x_{f,N}|f \in \varphi_{\mathcal{G}}\}$ form an additive set, then we have

$$\begin{aligned}\mu^p(\Gamma) &= \sum_{f \in \varphi_{\mathcal{G}}} \mu^p(x_{f,N}) \\ &\geq \sum_{f \in G} h(x_{f,N}) \ (\ Lemma\ 5)\end{aligned}$$

According to equation (7.1), we see that $\mu^p(\Gamma) \geq h(\Gamma)$. Since $h^r(\Gamma)$ is the minimum $\mu^p(\Gamma)$ over all solution plans, we have $h^r(\Gamma) = \min_p \mu^p(\Gamma) \geq h(\Gamma)$. □

Theorem 4 shows that $h(\Gamma)$ is indeed a lower bound of the pending cost. Hence, during the search, we can use $g(\Gamma) + h(\Gamma)$ as a admissible heuristic to guide the search.

Algorithm 10: cost_init()

Input: $\Psi^s = (\mathcal{F}^s, \mathcal{A}^s, \varphi_I, \varphi_G)$, $\Phi^c = (V, C, \mu)$, N

1 **for** *all* $x_{f,0} \in V$ **do**
2   set $h(x_{f,0}) = 0$ if $f \in \varphi_{\mathcal{I}}$ and $h(x_{f,0}) = \infty$ otherwise ;
3 **for** *t=0 to* N **do**
4   **for** *all* $x_{a,t} \in V$ **do**
5   compute $h(x_{a,t})$ using Definition 30;
6   **for** *all* $x_{f,t+1} \in V$ **do**
7   compute $h(x_{f,t+1})$ using Definition 30;

Implementation

The algorithms for initializing and maintaining the $h(x)$ values for all $x \in V$ are shown in Algorithms 10 and 11, respectively. To initialize the cost function $h(x)$, we first set $h(x_{f,0}) = 0$ if $f \in \varphi_{\mathcal{I}}$ and $h(x_{f,0}) = \infty$ otherwise. Then, we set the initial values for variables from time step 0 to N following Definition 30. In addition, for each action variable $x_{a,t}$, we pre-compute whether its precondition variables are additive. We also pre-compute whether all the goal variables are additive. Then we decide whether to use max or $\sum$, according to Definition 30 and Equation (7.1).

Algorithm 11 updates the h values if no conflict occurs during the search. It uses a priority queue Q to store all variables whose h values need to be updated after a constraint propagation. Since the variables in Q are ordered by the time step t, the variables will be updated in an increasing order of t. When $h(x)$ values are properly maintained, $h(\Gamma)$ for any partial assignment Γ can be computed easily using Equation (7.1). The updated $h(\Gamma)$ will be used in Line 17 of the BB-DPLL() procedure. For the example in Figure 7.1, if $x_{a_1,1}$ is assigned a value, then the h values of $x_{f_4,2}, x_{a_3,2}, x_{a_4,2}, x_{f_6,3}$ and $x_{f_7,3}$ will be updated.

7.3.2 Action Cost Based Variable Branching

In the BB-DPLL procedure, the variable branching scheme is the same to that in MiniSat [134, 34], a variant of VSIDS [88], which works as follows:

1. Each variable x has a priority value $p(x)$, initialized to 0. δ_p is a priority increment that is initialized to 1.

Algorithm 11: cost_propagate()

Input: $\Psi^s = (\mathcal{F}^s, \mathcal{A}^s, \varphi_I, \varphi_G)$, $\Phi^c = (V, C, \mu)$

```
1  initialize Q as a priority queue sorted by t;
2  while Q ≠ ∅ do
3      get x from Q, Q ← U\{x};
4      if x = x_{a,t} ∈ V then
5          if Γ(x_{a,t})=false then newcost ← ∞;
6          else
7              compute newcost using Definition 30;
8          if newcost ≠ h(x_{a,t}) then
9              h(x_{a,t}) ← newcost;
10             for all f ∈ add(a) do
11                 U ← U ∪ {x_{f,t+1}};
12     else if x = x_{f,t} ∈ V then
13         if Γ(x_{f,t})=false then newcost ← ∞;
14         else compute newcost using Definition 30;
15         if newcost ≠ h(x_{f,t}) then
16             h(x_{f,t}) ← newcost;
17             for all a such that f ∈ pre(a) do
18                 Q ← Q ∪ {x_{a,t}};
```

2. In decide(), with a constant probability P_0, randomly select an unassigned variable x, and with probability $1 - P_0$, select the unassigned variable with the highest priority value. Assign the selected variable to $true$.

3. Whenever a learnt clause is generated by analyze() in BB-DPLL, for each variable x in the new learnt clause, we update the priority values $p(x)$ by $p(x) = p(x)+\delta_p$. After that, multiply δ_p by a constant $\theta > 1$.

4. Periodically divide all priority values by a large constant γ and reset δ_p to 1.

VSIDS is widely used in most SAT algorithms. We present a planning specialized branching scheme that is more effective. Essentially, we use an estimated total action costs when branching. The rules are as follows:

1. Each variable x has a priority value $p(x)$. Initialize $p(x)$ as follows:

$$p(x) = \begin{cases} \mu(a), & \text{if } x = x_{a,t} \in V \\ 0, & otherwise \end{cases}$$

2. δ_p is a priority increment that is initialized to 1.

3. In decide(), with a constant probability P_0, randomly select an unassigned variable x, and with probability $1 - P_0$, select the unassigned variable with the highest priority value. Assign the selected variable to $false$.

4. Whenever a learnt clause is generated by analyze() in BB-DPLL, for each variable x in the new learnt clause, update the priority values $p(x)$ as follows:

$$p(x) = \begin{cases} p(x) & + & \mu(a)\delta_p, & \text{if } x = x_{a,t} \in V \\ p(x) & + & \delta_p, & \text{otherwise} \end{cases}$$

   After that, multiply δ_p by a constant $\theta > 1$.

5. Periodically divide all priority values by a large constant γ and reset δ_p to 1.

Note that we keep the parameters setting exactly the same to MiniSat. That is, $P_0 = 0.02$, $\theta_p = 1.2$, and $\gamma = 100$. Comparing with the VSIDS heuristic, our BB-DPLL heuristic is often a better estimation, which gives higher priorities to variables with higher costs.

7.4 Experimental Results

We test seven temporal expressive planners. Sun Java 1.6 and Python 2.6 run-time systems are used. The time limit, for each instance, is set to 3600 seconds. We have four domains in this experiments, which are the same to the instances in Chapter 6.

Seven planners are tested and compared. We test four temporally expressive planners: Crikey2 [23], Crikey3 [24] (statically-linked binary for x86 Linux), LPG-c [48] and Temporal Fast Downward (TFD) [38]. SCP is the basic SCP framework without minimizing action costs, which in fact the same to PET that is introduced in Chapter 6. We use a different name here because to compare with our planning specialized algorithm. The SAT solver is changed to MiniSAT2 [34]. SCP does not optimize total action costs. The planners that use the methods that we proposed are called SCP^{max} (Section 7.2) and SCP^{bb} (Section 7.3), respectively. SAT4J [113], the winner of weighted partial Max-SAT (industrial track) in the Max-SAT 2009 Competition [125] is used in SCP^{max}.

P	Crikey3			SCP			SCP^{max}		SCP^{bb}	
	T	H	C	T	H	C	T	C	T	C
1	0.1	22	4110	4.9	22	1980	21.5	520	60.4	520
2	0.1	32	5190	19.1	32	3560	233.2	800	157.5	900
3	0.2	40	7340	134.0	40	6000	514.0	1830	1156.8	1160
4	1.4	72	12110	4.2	27	2000	15.2	1320	12.0	1320
5	7.2	100	20190	10.5	34	3310	76.4	2200	253.3	2320
6	111.5	150	30290	22.5	39	4650	618.3	3300	193.2	3550
7		TLE		61.6	54	6360	3513.6	4300	1121.9	4550
8	287.7	200	35340	122.1	49	5430	2624.2	4060	3475.4	4180
9		TLE		662.9	60	6960	2340.7	5470	454.8	6120
10		TLE		757.8	32	4010	388.2	3610	228.2	3500
11		TLE		45.1	20	3200	210.9	2700	58.2	2700
12		TLE		88.9	23	4190	162.4	3610	114.0	3600
13		TLE		1675.4	31	5370	2714.2	4720	1056.8	4500
14		TLE		3303.0	29	6770	2141.8	5720	2788.0	5600
Σ		n/a		6911.9	492	63790	15574.6	44160	11130.3	44520

Table 7.1: Experimental results in the P2P domain. Column 'P' is the instance ID. Columns 'T', 'H' and 'C' are the solving time, makespan and total action costs of solutions, respectively.

The P2P domain

We first experiment on a collection of instances in the P2P domain. The results are shown in Table 7.1. If a planner is able to solve every instance, we present in Column 'Σ' the summation of the solving time, makespan, and total action costs over all instances. 'Timeout' means that the solver runs out of the time limit of 3600s and '-' means no solution is found. This is for an easier comparison of different metrics between the solvers. Crikey2, LPG-c and TFD are not included because they all fail to solve any instance. We do not show makespan ("H") for SCP^{bb} and SCP^{max} since they give the same makespans as SCP.

Instances 1 to 9 have simple topologies. Crikey2 fails to solve any instance in this category. Crikey3 solves 7 out of 14 instances. It is faster on two simpler instances but slower than SCP on two other larger instances. Overall, the makespans of solutions found by Crikey3 are up to five times longer than those found by SCP. SCP also outperforms Crikey3 by up to 7 times in terms of the total action costs. Instances 10 to 14 have more complicated network topologies. Both Crikey2 and Crikey3 fail to solve any of these instances. Crikey3 times out and Crikey2 reports no solution found. It may be due to their incompleteness.

SCP, SCP^{bb} and SCP^{max} solve all the instances. In general, SCP^{bb} runs longer than the other two. The total action costs by SCP^{max} and SCP^{bb} are consistently lower (by up to 4 times) than that of SCP. Overall, SCP^{bb} has comparable, if not better, performance than SCP^{max} in this domain; the former has a slightly worse total action costs (0.8% more) but much shorter solving time (28% shorter).

P	Crikey2			Crikey3			LPG-c			TFD			SCP			SCP^{max}		SCP^{bb}	
	T	H	C	T	H	C	T	H	C	T	H	C	T	H	C	T	C	T	C
1	3.0	13	332	0.1	18	312	0.1	17	352	0.0	13	332	2.6	13	1052	5.2	332	2.6	332
2	0.7	11	150	0.3	14	130	12.6	9	160	0.1	9	110	1.8	9	900	3.1	110	1.8	110
3	2.9	23	352	0.1	28	332		-		0.0	23	352	6.8	22	2032	16.0	352	8.0	352
4	9.6	19	452	0.1	34	432		-		0.0	19	452	7.2	18	1792	15.0	452	7.3	452
5	52.8	35	644	0.1	43	524		-		0.0	25	644	13.2	24	1664	29.9	644	24.1	644
6	24.4	39	906	1.4	47	584		-		8.3	34	1096	20.6	25	1936	49.7	566	42.8	566
7	131.9	37	804	0.4	58	684		-		1.2	42	1008	88.3	31	2316	195.5	704	241.3	704
8	73.8	42	854	1.9	58	734		-		20.8	35	1154	131.0	30	2566	400.3	856	307.6	856
9	60.4	39	644	0.1	43	624		-		7.3	39	1144	26.7	26	1884	65.9	844	68.6	844
10	234.1	28	936	0.1	58	714		-		0.0	30	734	55.6	28	2144	109.4	734	162.4	734
11	376.7	47	1005	0.7	58	684		-		0.0	33	804	152.0	31	2334	265.7	804	413.0	804
Σ	970.4	333	7079	5.3	459	5754		-		37.8	292	7830	505.9	257	20620	1155.6	6398	1279.4	6398

Table 7.2: Experimental results on the Matchlift domain.

The Matchlift domain

The results on the Matchlift domain are in Table 7.2. On all instances, Crikey3 is the fastest, but with the lowest quality in terms of makespan. Surprisingly, Crikey3 finds the solutions with minimum action costs because for this problem domain lower costs can be achieved under a longer makespan. TFD is the second fastest. The solution quality by TFD is comparable to Crikey2, with slightly larger action cost but shorter makespan.

SCP is the second fastest, with the optimal makespans, but higher total action costs than Crikey3. Crikey2 is slightly faster, but also has worse makespan and total action costs than SCP^{max} and SCP^{bb}. The solutions by SCP^{max} and SCP^{bb} have comparable speed and exactly the same quality. They give optimal makespan and only slightly worse action costs. LPG-c is the worst in this domain, with only two instances solved.

The Matchlift-Variant domain

As shown by Table 7.3, SCP finds optimal makespan on all instances tested, whereas Crikey2 and Crikey3 run out of time on most instances and generate suboptimal plans on a few instances they finish. For the instances they solve, Crikey3 has the worst makespan. Except on very small instances, we can see that SCP not only finds the optimal solutions, but also runs faster than Crikey2 and Crikey3. Both LPG-c and TFD are not good in this domain, with only three instances solved.

In this domain, SCP^{bb} runs slightly slower than SCP, but finds solutions with significantly lower total action costs than the latter. Although on certain instances SCP^{max} can find solutions with exactly the same quality, it runs up to six times slower than SCP^{bb} to reach those solutions.

P	Crikey2			Crikey3			LPG-c			TFD			SCP			SCP^{max}		SCP^{bb}	
	T	H	C	T	H	C	T	H	C	T	H	C	T	H	C	T	C	T	C
1	10.9	14	220	5.1	17	200	-			-			2.5	13	840	8.1	220	2.4	220
2	147.7	13	374	7.5	16	334	0.4	13	354	-			1.7	13	1294	4.2	254	1.7	254
3	6.1	19	392	0.1	23	372	-			0.5	19	414	5.7	18	1336	12.3	392	5.7	392
4	106.0	25	444	41.8	27	424	460.9	21	354	-			6.9	21	1624	22.8	342	6.7	342
5	20.3	23	482	0.1	33	462	-			5.9	29	492	9.4	22	1962	23.1	482	16.0	482
6	121.6	25	464	42.0	27	444	0.1	33	362	-			5.9	21	1454	22.7	362	7.2	362
7	-			TLE			-			-			9.9	16	1636	37.1	396	10.0	396
8	167.1	17	290	TLE			-			-			53.3	16	1484	292.7	290	52.8	290
9	TLE			TLE			-			104.7	33	734	22.8	22	1668	2582.7	534	52.5	534
10	TLE			TLE			-			-			94.3	20	1726	488.7	416	94.6	416
11	TLE			TLE			-			-			361.4	16	1398	543.2	476	366.1	476
12	TLE			TLE			-			-			3.1	13	1008	8.9	358	2.3	358
Σ	n/a			n/a			n/a			n/a			576.9	211	17430	4046.3	4522	618.1	4522

Table 7.3: Experimental results in the Matchlift-Variant domain. 'TLE' means that the solver runs out of the time limit of 3600s and '-' means no solution is found.

P	Crikey2			Crikey3			LPG-c			SCP			SCP^{max}		SCP^{bb}	
	T	H	C	T	H	C	T	H	C	T	H	C	T	C	T	C
1	16.7*	122	2850	0.1	224	2600	336.7	712	4650	182.7	102	2660	306.4	2700	216.0	1760
2	4.0	122	1450	0.1	122	1400	2.2	244	2750	202.6	122	2670	500.0	1450	551.8	775
3	18.8	122	4450	0.1	225	3200	712.5	346	4900	232.1	122	4505	451.5	3900	231.8	1935
4	19.8	122	3950	0.2	323	3700	365.3	122	4600	233.9	122	3575	450.0	4050	233.3	1935
5	38.3	102	3900	0.1	238	3600	653.0	224	4100	241.7	102	1940	589.7	3900	629.4	2010
6	10.4*	122	2700	0.1	326	2500	85.8	224	4300	217.8	118	6020	402.1	3700	218.7	1930
7	201.4	102	6400	0.2	102	4700	9.3	102	5000	432.5	102	3330	693.2	5800	422.6	3020
8	180.4	102	7500	0.2	125	5050	2.2	102	5150	443.5	102	7330	678.6	5900	423.8	2530
9	159.5*	102	6900	0.2	125	5050	15.9	102	4600	443.5	102	4940	677.3	6100	434.0	3080
Σ	649.3	1018	40100	1.3	1810	31800	2183.01	2178	40050	2630.4	994	36970	4748.8	37500	3361.3	18975

Table 7.4: Experimental results in the Driverslogshift domain. The result marked with a '*' means that the solution is invalid.

The Driverslogshift domain

Crikey3 again is the fastest among all planners. Its makespans, however, are much worse than that of all others. As shown in Table 7.4, the optimal makespans of the instances tested, provided by SCP, SCP^{max} and SCP^{bb}, are typically much shorter than those by Crikey3. For example, the optimal makespan for Instance 4 in Table 7.4 is about one third of the makespan reported by Crikey3. LPG-c finds solutions with solving times that are similar to SCP's, which is much slower than Crikey2 and Crikey3. Its solution quality is however the worst: its makespans are comparable to Crikey3, and the total action costs are about the same to Crikey2. TFD is not included because it fails to solve any instance in this domain.

In terms of action costs, SCP^{bb} is by far the best, much better than the plans found by Crikey2, Crikey3, SCP, and SCP^{max}, have similar quality. In this domain, SCP^{bb} is also much more efficient than SCP^{max}.

7.5 Summary

We have extended the SAT-based temporally expressive planning approach to further handle action costs. Two approaches are studied: one is to compile CSTE tasks into Max-SAT problems and take advantage of existing Max-SAT solvers; the other is a planning specialized branch and bound algorithm. By comparing our approaches with the state-of-the-art planners, both approaches are very competitive not only in plan quality but also in performance. In particular, the BB-DPLL algorithm is competitive with, if not better than, the Max-SAT approach for SAT-based CSTE planning.

Chapter 8

Conclusions and Future Research

The contribution of this dissertation is devising new methods accommodating to the recent advances in planning formulations. The recent introduced SAS+ formulism induces interesting structures such as domain transition graph. We have developed an abstraction search method that conducts search directly on domain transition graphs. We also have recognized that the information enforced by domain transition graphs can be used as additional constraints to boost state-of-the-art SAT-based planners. The new encoding scheme SASE fully exploits the potentials of SAS+, and it greatly improves the efficiency of planning as SAT method. We study the search space induced by SASE from both theoretical and empirical perspectives. In particular, our study has explained and revealed why SASE is more efficient. Finally we extend the planning as Satisfiability approach to handle temporal and action costs.

Our study strongly suggests the advantage of applying SAS+ in planning algorithms. In particular, the state space induced by the transition make the resulting problem naturally have a hierarchy between transitions and actions. Various planning algorithms may benefit from capturing this property.

8.1 Future Works

Formulation is so significant in planning algorithms that our research may inspire new techniques in various fields. Below, we review some related works and discuss a few directions for future research.

8.1.1 Adaptive Search Strategies for Abstractions

DTG-Plan performs faster than the state-of-the-art with less memory consumption. Being greedy is however a double blade, which makes DTG-Plan fail in certain instances. Therefore, the incremental abstraction strategy needs improvements, although in theory it is complete. In the high level, the algorithm for extending the abstractions is still not flexible enough. Right now our approach is straightforward. We randomly pick those DTGs that are not included yet, but depended by the abstraction. If there are too many candidates, random decisions are made. We believe it is promising to have more guidances.

In addition, by having multiple layers of decisions, DTG-Plan leaves us great opportunities to devise better strategies. In each individual component, it is possible to have adaptive methods to help GIR make better decisions, when certain patterns in the historical data are observed. For example, while searching for a set of facts (i.e. search_fact_set()), we use heuristics such as forced ordering to pick facts. Nevertheless, as a heuristic, it may not always make correct decisions. If some ordering always lead to a dead-end, it could be reasonable to override current ordering and try with alternatives.

8.1.2 Other Semantics and Encodings

Many enhancements have been developed for SAT based planning since it was first introduced [71]. The split action representation [71, 35] uses a conjunction of multiple variables to represent an action. The optimality of parallelism is however lost. Robinson proposes a new way of doing splitting without sacrificing the optimality [109]. The results show that this method has advantages and improvements over SatPlan06 [74]. There are many literatures with thorough analysis, or improvements along this line of research on SatPlan family encodings. In particular, the power of mutual exclusion, in the context of planning as SAT, has attracted interests [20, 116]. A new encoding scheme called SMP is proposed, which preserves the merits of londex and shows certain advantages over exiting encoding schemes [116].

The research along the line of SatPlan are all for optimal makespans. This makespan optimal semantics, along with another parallel semantics, are studied and formalized as $\forall$-step and $\exists$-step, respectively [30, 107]. $\exists$-step enforces weaker mutual exclusions than $\forall$-step does, thus may lead to faster overall problem solving by having fewer rounds of SAT tests. As a trade-off, it loses the optimality of time steps. The semantics in both SatPlan06 and SASE are $\forall$-step. The research on various kinds of semantics are orthogonal to our contribution in SASE, and the idea of SASE can be migrated to new semantics.

Since SAT based planning is also applied in sequential planning, we believe the idea of SASE can also be extended to this field. The first planner of this kind is MEDIC [35]. It implements the idea of splitted action representation, combined with several other new methods such as those handling the frame axioms. This study shows that in the case of sequential planning, splitting yields very competitive encodings. Some research also proposes to utilize the advantages of both sequential and parallel planning [17]. As an anytime algorithm, it optimizes the number of actions in addition to makespans.

8.1.3 Additional Techniques for Planning as SAT

The tremendous amount of two-literal clauses (such as those mutual exclusion clauses in the case of planning) is a key challenge to approaches based on satisfiability tests. Some research proposed to mitigate the burden of encoding them by recognizing certain structures [12]. In traditional SAT planning systems like SatPlan06, the mutual exclusions are encoded in a quadratic manner. Rintanen proposes a log size technique [104], called clique representation, for the mutual exclusion constraints, and later a linear size one [107]. The mutual exclusions in SASE are represented by the clique representation. The log size of the clique representation is supposed to be less compact than the linear encoding. Our results, however, have shown that SASE is in general more compact. We believe the major reason is the compactness of SAS+ formalism. It is, certainly, an interesting open question whether the linear size encoding technique can be applied to further improve SASE.

There are also techniques beyond encoding to boost SAT-based planning. Rintanen introduces how to incorporate symmetry information into SAT instances [103]. MaxPlan [131] does a planning graph analysis to find an upper bound on the optimal make span and then does SAT queries using decreasing time steps, until it meets a unsatisfiable SAT instance. A lemma reusing method is proposed in [91] to reuse the learnt clauses across multiple SAT solvings. In [99], a multiple-step query strategy is introduced, which however asks for modified SAT procedures. In this method, only a single call to a SAT solver is needed to handle multiple time steps. All these boosting methods are proposed in the context of STRIPS based planning. Therefore, it is interesting to incorporate them into SASE and study if they can further improve the performance of SASE.

8.1.4 More Understanding of Structures in General SAT Instances

As one of the most intensively studied problems in computer science, SAT is hard from a complexity perspective. Modern solvers, however, are quickly improving their efficiency. It is therefore

interesting and important to understand why SAT solvers work so well on certain instances, and furthermore, what makes a SAT instance easy or hard. There is much prior research that tries to make such understanding, including backdoor set [130] and backbone [87]. Backdoor set variables are a set of variables, such that when these variables are assigned, other variables' assignments can be derived in polynomial time. Backbone variables are those variables that have the same assignment in all valid solutions.

It will be interesting to see if there are connections between SASE's problem structure and those proposed theories in the literature. For example, is the improvement of SASE because it can lead to a smaller backdoor set? Second, we have shown that the efficiency of SASE is a result of transition variables' significance, and there is strong correlation between the speedup from SASE and the transition index. It is interesting to investigate if similar variable set and predictive index can be automatically found for general SAT solving.

8.1.5 SAS+ in More Advanced and Expressive Planning Models

As mentioned in Chapter 6, the SAT-based temporally expressive planning approach has the clear advantage that it handles concurrency in a stronger form. It however does not support numerical durations. Theoretically, any temporal information can always be compiled into a discrete space, thus this limitation is easy to overcome. In that case, our approach can handle durations of real values, as long as we can recompile the unit using a smaller unit. Nevertheless, doing this way may lead to unnecessarily large planning task formulas. Besides using explicit time for temporal information, there are also partial order style models [1] in temporal reasoning research. These could be an alternative to support numerical durations.

Another limitation is that we assume action durations are constants. To resolve this might be more difficult. We believe it is possible to have multiple copies of the same action, each has a distinct duration. By doing this, the resulted encoding again might be very large.

In addition to the future research directions discussed above, given the efficiency of SASE, it is promising to apply it to other SAT-based planning approaches, such as those for complex planning with preferences [50] and temporal features [64]. On the other hand, given the strong connection between planning as SAT and planning as CSP, the ideas in SASE can be migrated to those CSP based planning approaches.

References

[1] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26:832–843, 1983.

[2] F. Bacchus. The power of modelling - a response to PDDL2.1. *Journal of Artificial Intelligence Research*, 20:125–132, 2003.

[3] F. Bacchus and M. A. Winter. Planning with resources and concurrency a forward chaining approach. In *Proceedings of International Joint Conference on Artificial Intelligence*, 2001.

[4] C. Bäckström and B. Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11:625–655, 1996.

[5] D. Bauer and A. Koller. Sentence Generation as Planning with Probabilistic LTAG. In *Proc. of Int'l Conference on Tree Adjoining Grammars and Related Formalisms*, 2010.

[6] A. R. Benaskeur, F. Kabanza, and E. Beaudry. CORALS:a real-time planner for anti-air defense operations. *ACM Transactions on Intelligent Systems and Technology*, 1(2):1–20, 2010.

[7] A. Bhattacharya and S. Ghosh. Self-optimizing peer-to-peer networks with selfish processes. In *Proceedings of International Conference on Self-Adaptive and Self-Organizing Systems*, 2007.

[8] A. Biere. Pr{e,i}coSAT. In *SAT'09 Competition*, 2009.

[9] B. Blai and G. Hector. Planning as heuristic search. *Artificial Intelligence*, 129:5–33, 2001.

[10] A. Blum and M. Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, 90:1636–1642, 1997.

[11] B. Bonet and H. Geffner. Planning as Heuristic Search. In *Artificial Intelligence*, 2001.

[12] R. I. Brafman. A simplifier for propositional formulas with many binary clauses. In *Proceedings of International Joint Conference on Artificial Intelligence*, 2001.

[13] L. Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

[14] M. Briel, T. Vossen, and S. Kambhampati. Reviving Integer Programming Approaches for AI Planning. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2005.

[15] D. Bryce, M. Verdicchio, and S. Kim. Planning interventions in biological networks. *ACM Transactions on Intelligent Systems and Technology*, 1(2), 2010.

[16] E. Burns, S. Lemons, R. Zhou, and W. Ruml. Best-First Heuristic Search for Multi-Core Machines. In *Proceedings of International Joint Conference on Artificial Intelligence*, 2009.

[17] M. Büttner and J. Rintanen. Satisfiability Planning with Constraints on the Number of Actions. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2005.

[18] M. Carman, L. Serafini, and P. Traverso. Web service composition as planning. In *ICAPS Workshop on Planning for Web Services*, 2003.

[19] C. Castellini, E. Giunchiglia, and A. Tacchella. SAT-based planning in complex domains: concurrency, constraints and nondeterminism. *Artificial Intelligence*, 147:85–117, 2003.

[20] Y. Chen, R. Huang, Z. Xing, and W. Zhang. Long-distance mutual exclusion for planning. *Artificial Intelligence*, 173:197–412, 2009.

[21] Y. Chen, R. Huang, and W. Zhang. Fast Planning by Search in Domain Transition Graphs. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2008.

[22] M. Cirillo, L. Karlsson, and A. Saffiotti. Human-aware task planning: an application to mobile robots. *ACM Transactions on Intelligent Systems and Technology*, 1(2), 2010.

[23] A. Coles, M. Fox, K. Halsey, D. Long, and A. Smith. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence*, 173:1–44, 2008.

[24] A. Coles, M. Fox, D. Long, and A. Smith. Planning with problems requiring temporal coordination. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2008.

[25] S. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[26] W. Cushing, S. Kambhampati, Mausam, and D. S. Weld. When is temporal planning really temporal? In *Proceedings of International Joint Conference on Artificial Intelligence*, 2007.

[27] W. Cushing, S. Kambhampati, K. Talamadupula, D. S. Weld, and Mausam. Evaluating temporal planning domains. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2007.

[28] S. Darras, G. Dequen, L. Devendeville, and C. Li. On inconsistent clause-subsets for Max-SAT solving. In *Proceedings of International Joint Conference on Principles and practice of constraint programming*, pages 225–240, 2007.

[29] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[30] Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*, pages 169–181. Springer-Verlag, 1997.

[31] M. Do and S. Kambhampati. SAPA: A Multi-objective Metric Temporal Planner. *Journal of Artificial Intelligence Research*, 20:155–194, 2003.

[32] M. B. Do, W. Ruml, and R. Zhou. On-line planning and scheduling: An application to controlling modular printers. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2008.

[33] S. Edelkamp and M. Helmert. MIPS: The model-checking integrated planning system. *AI Magazine*, 22, 2001.

[34] N. Een and N. Sörensson. An Extensible SAT-solver, 2003.

[35] M. Ernst, T. Millstein, and D. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of International Joint Conference on Artificial Intelligence*, 1997.

[36] K. Erol, J. Hendler, and D. S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of AAAI Conference on Artificial Intelligence*, 1995.

[37] The Fifth Max-SAT Evaluation. http://maxsat.ia.udl.cat/rules/, 2010.

[38] P. Eyerich, R. Mattmüller, and G. Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2009.

[39] M. Fox and D. Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367421, 1998.

[40] M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of International Joint Conference on Artificial Intelligence*, 1999.

[41] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

[42] E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.

[43] Z. Fu and S. Malik. On solving the partial Max-SAT problem. In *Proceedings of Theory and Applications of Satisfiability Testing*, pages 252–265, 2006.

[44] Z. Fu and S. Malik. Solving the minimum-cost satisfiability problem using SAT based branch-and-bound search. In *Proceedings of International Joint Conference on Computer-aided design*, pages 852–859, 2006.

[45] M. Gavanelli. The log-support encoding of CSP into SAT. In *Proceedings of Principles and Practice of Constraint Programming*, pages 815–822, 2007.

[46] A. Gerevini, A. Saetti, and I. Serina. An approach to efficient planning with numerical fluents and multi-criteria plan quality. *Artificial Intelligence*, 172(8-9):899–944, 2008.

[47] A. Gerevini, A. Saetti, and I. Serina. Temporal planning with problems requiring concurrency through action graphs and local search. *Proceedings of International Conference on Automated Planning and Scheduling*, 2010.

[48] A. Gerevini and I. Serina. LPG: a planner based on local search for planning graphs with action costs. In *Proc. of the Sixth Int. Conf. on AI Planning and Scheduling*, pages 12–22. Morgan Kaufman, 2002.

[49] M. Ghallab, D. S. Nau, and P. Traverso. *Automated Planning*. Morgan Kaufmann, 2004.

[50] E. Giunchiglia and M. Maratea. Planning as satisfiability with preferences. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2007.

[51] P. Hansen and B. Jaumard. Algorithm for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.

[52] P. Haslum. Reducing accidental complexity in planning problems. In *Proceedings of International Joint Conference on Artificial Intelligence*, 2007.

[53] P. Haslum and H. Geffner. Heuristic planning with time and resources. In *Proc. IJCAI-01 Workshop on Planning with Resources*, 2001.

[54] M. Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[55] M. Helmert. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, pages 503–535, 2008.

[56] M. Helmert and C. Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of International Conference on Automated Planning and Scheduling*, 2009.

[57] M. Helmert, P. Haslum, and J. Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2007.

[58] M. Helmert, P. Haslum, and J. Hoffmann. Explicit-State Abstraction: A New Method for Generating Heuristic Functions. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2008.

[59] M. Helmert and G. Röger. How Good is Almost Perfect? In *Proceedings of AAAI Conference on Artificial Intelligence*, 2008.

[60] J. Hoffman. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.

[61] J. Hoffmann, H. Kautz, C. Gomes, and B. Selman. SAT encodings of state-space reachability problems in numeric domains. In *Proceedings of International Joint Conference on Artificial Intelligence*, 2007.

[62] J. Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[63] Y. Hu. Temporally-expressive planning as constraint satisfaction problems. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2007.

[64] R. Huang, Y. Chen, and W. Zhang. An Optimal Temporally Expressive Planner: Initial Results and Application to P2P Network Optimization. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2009.

[65] R. Huang, Y. Chen, and W. Zhang. A novel transition based encoding scheme for planning as satisfiability. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2010.

[66] P. Jonsson and C. Bäckström. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100(1-2):125–176, 1998.

[67] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. 1972.

[68] M. Katz and C. Domshlak. Optimal admissible composition of abstraction heuristics. *Artificial Intelligence*, 174:767–798, 2010.

[69] H. Kautz. SATPLAN04: Planning as Satisfiability. In *Abstracts IPC4*, 2004.

[70] H. Kautz, D. McAllester, and B. Selman. Encoding plans in propositional logic. In *Proceedings of Principles and Knowledge Representation and Reasoning*, 1997.

[71] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of European Conference on Artificial Intelligence*, 1992.

[72] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of AAAI Conference on Artificial Intelligence*, 1996.

[73] H. Kautz, B. Selman, and J. Hoffmann. Unifying sat-based and graph-based planning. In *Proceedings of International Joint Conference on Artificial Intelligence*, 1999.

[74] H. Kautz, B. Selman, and J. Hoffmann. SatPlan: Planning as Satisfiability. In *Abstracts IPC5*, 2006.

[75] A. Kishimoto, A. Fukunaga, and A. Botea. Scalable, Parallel Best-First Search for Optimal Sequential Planning. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2009.

[76] C. A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.

[77] J. Koehler and J. Hoffmann. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research*, 12:338–386, 2000.

[78] A. Koller and M. Stone. Sentence Generation as a Planning Problem. In *Annual Meeting of the Association for Computational Linguistics*, 2007.

[79] J. Kvarnström and M. Magnusson. TALplanner in IPC-2002: Extensions and control rules. *Journal of Artificial Intelligence Research*, 20:343–377, 2003.

[80] J. Larrosa, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Branch and bound for boolean optimization and the generation of optimality certificates. In *Proceedings of International Conference on Theory and Applications of Satisfiability Testing*, pages 453–466. Springer-Verlag, 2009.

[81] C. Li, F. Manya, N. Mohamedou, and J. Planes. Exploiting cycle structures in Max-SAT. In *Proceedings of 12th International Joint Conference on the Theory and Applications of Satisfiability Testing*, pages 467–480, 2009.

[82] C. Li, F. Manya, and J. Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30:321–359, 2007.

[83] X. Y. Li. Optimization algorithms for the minimum-cost satisfiability problem. *PhD Thesis, Department of Computer Science, North Carolina State University, North Carolina*, 2004.

[84] D. Long and M. Fox. Exploiting a graphplan framework in temporal planning. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2003.

[85] R. Mattmüller and J. Rintanen. Planning for temporally extended goals as propositional satisfiability. In *Proceedings of International Joint Conference on Artificial Intelligence*, 2007.

[86] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL:the planning domain definition language. Technical Report TR-98-003, Yale University, 1998.

[87] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic ‘phase transitions’. *Nature*, 400(8):133–137, 1999.

[88] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, 2001.

[89] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of Design Automation Conference*, 2001.

[90] J. L. Myers and A. D. Well. *Research Design and Statistical Analysis*. Routledge, 2nd edition, 2003.

[91] H. Nabeshima, T. Soh, K. Inoue, and K. Iwanuma. Lemma reusing for SAT based planning and scheduling. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2006.

[92] X. Nguyen and S. Kambhampati. Extracting effective and admissible heuristics from the planning graph. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2000.

[93] X. Nguyen, S. Kambhampati, and R. S. Nigenda. Planning graph as the basis for deriving heuristics for plan synthesis by state space and csp search. In *Artificial Intelligence*, volume 135(12), pages 73–123, 2002.

[94] J. Penberthy and D. Weld. Temporal planning with continuous change. In *Proceedings of AAAI Conference on Artificial Intelligence*, pages 1010–1015, 1994.

[95] D. Pham, J. Thornton, and A. Sattar. Modelling and solving temporal reasoning as propositional satisfiability. *Artificial Intelligence*, 172:1752–1782, 2008.

[96] K. Pipatsrisawat and A. Darwiche. Clone: solving weighted Max-SAT in a reduced search space. In *Proceedings of the 20th Australian joint conference on Advances in artificial intelligence*, pages 223–233, 2007.

[97] J. Porteous, M. Cavazza, and F. Charles. Applying planning to interactive storytelling: Narrative control using state constraints. *ACM Transactions on Intelligent Systems and Technology*, 1(2):111–130, 2010.

[98] J. Rao and X. Su. A survey of automated web service composition methods. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition*, pages 43–54, 2004.

[99] K. Ray and M. L. Ginsberg. The complexity of optimal planning and a more efficient method for finding solutions. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2008.

[100] I. Refanidis and N. Yorke-Smith. A constraint based approach to scheduling an individual's activites. *ACM Transactions on Intelligent Systems and Technology*, 1(2), 2010.

[101] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[102] S. Richter, M. Helmert, and M. Westphal. Landmarks Revisited. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2008.

[103] J. Rintanen. Symmetry Reduction for SAT Representations of Transition System. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2003.

[104] J. Rintanen. Biclique-based representations of binary constraints for making SAT planning applicable to larger problems. In *Proceedings of European Conference on Artificial Intelligence*, 2006.

[105] J. Rintanen. Complexity of concurrent temporal planning. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2007.

[106] J. Rintanen. Complexity of concurrent temporal planning. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2007.

[107] J. Rintanen, K. Heljanko, and I. Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 12-13:1031–1080, 2006.

[108] N. Robinson, C. Gretton, D. Pham, and A. Sattar. A compact and efficient sat encoding for planning. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2008.

[109] N. Robinson, C. Gretton, D. Pham, and A. Sattar. SAT-Based Parallel Planning Using a Split Representation of Actions. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2009.

[110] N. Robinson, C. Gretton, D. N. Pham, and A. Sattar. Cost-optimal planning using weighted MaxSAT. In *ICAPS workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, 2010.

[111] A. Rudenko. *Automated Planning for Open Network Architectures*. PhD thesis, UCLA, 2002.

[112] W. Ruml, M. B. Do, and M. P. J. Fromherz. On-line planning and scheduling for high-speed manufacturing. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2005.

[113] SAT4J solver. http://www.sat4j.org/, 2004.

[114] B. Selman, H. Kautz, and D. McAllester. Ten Challenges in Propositional Reasoning and Search. In *Proceedings of International Joint Conference on Artificial Intelligence*, 1997.

[115] J. Shin and E. Davis. Continuous time in a SAT-based planner. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2004.

[116] A. Sideris and Y. Dimopoulos. Constraint propagation in propositional planning. In *Proceedings of International Conference on Automated Planning and Scheduling*, 2010.

[117] D. Smith. The case for durative actions: A commentary on PDDL2.1. *Journal of Artificial Intelligence Research*, 20:149–154, 2003.

[118] D. Smith and D. Weld. Temporal planning with mutual exclusion reasoning. In *Proceedings of International Joint Conference on Artificial Intelligence*, 1999.

[119] M. Soos, K. Nohl, and C. Castelluccia. Extending sat solvers to cryptographic problems. In *International Conference on Theory and Applications of Satisfiability Testing*, 2009.

[120] K. Talamadupula, J. Benten, S. Kambhampati, P. Schermerhorn, and M. Scheutz. Planning for human-robot teaming in open worlds. *ACM Transactions on Intelligent Systems and Technology*, 1(2), 2010.

[121] The 5th International Planning Competition. http://zeus.ing.unibs.it/ipc-5/, 2006.

[122] The 6th International Planning Competition. http://ipc.informatik.uni-freiburg.de/homepage, 2008.

[123] The Fifth International Planning Competition. http://zeus.ing.unibs.it/ipc-5/, 2006.

[124] The Fourth International Planning Competition. http://ls5-www.cs.uni-dortmund.de/∼edelkamp/ipc-4/, 2004.

[125] The Fourth Max-SAT Evaluation. http://maxsat.ia.udl.cat/09/, 2009.

[126] The Sixth International Planning Competition. http://ipc.informatik.uni-freiburg.de/, 2008.

[127] The Third International Planning Competition. http://planning.cis.strath.ac.uk/competition/, 2002.

[128] V. Vidal and H. Geffner. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence*, 170:98–335, 2006.

[129] B. W. Wah and Y. Chen. Constraint partitioning in penalty formulations for solving temporal planning problems. *Artificial Intelligence*, 170:187–231, 2006.

[130] R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of International Joint Conference on Artificial Intelligence*, 2003.

[131] Z. Xing, Y. Chen, and W. Zhang. MaxPlan: Optimal Planning by Decomposed Satisfiability and Backward Reduction. In *5th International Planning Competition Booklet, International Conference on Automated Planning and Scheduling*, 2006.

[132] Z. Xing and W. Zhang. Maxsolver: An efficient exact algorithm for (weighted) maximum satisfiability. *Artificial Intelligence*, 164:47–80, 2005.

[133] H. L. S. Younes and R. G. Simmons. Vhpop: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, 20:405–430, 2003.

[134] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of the 14th Internatioal conference on computer-aided verification*, volume 2404, pages 17–36. Springer, 2002.

Vita

Ruoyun Huang

Date of Birth June 29, 1982

Place of Birth Yixing, China

Degrees Ph.D. Washington University in St. Louis, August 2011
B.S. Donghua University, Computer Science, May 2003

Publications

R. Huang, Y. Chen, and W. Zhang, "A Novel Transition Based Encoding Scheme for Planning as Satisfiability", Proceedings of AAAI Conference on AI (**AAAI'10**), 2010. **Outstanding Paper Award**

Y. Chen, *R. Huang*, Z. Xing, and W. Zhang, "Long-Distance Mutual Exclusion for Planning", **Artificial Intelligence Journal**, Volume 173, Issue 2, Pages 197-412, 2009.

R. Huang, Y. Chen, and W. Zhang, "An Optimal Temporally Expressive Planner: Initial Results and Application to P2P Network Optimization", Proceedings of International Conference on Automated Planning and Scheduling (**ICAPS'09**), 2009.

Y. Chen, *R. Huang*, and W. Zhang, "Fast Planning by Search in Domain Transition Graphs", Proceedings of AAAI Conference on AI (**AAAI'08**), 2008.

C. Hsu, B. Wah, *R. Huang*, and Y. Chen, "Constraint Partitioning for Solving Planning Problems with Trajectory Constraints and Goal Preferences", Proceedings of International Joint Conference on AI (**IJCAI'07**), 2007.

August 2011

Automated Planning Formulations, Huang, Ph.D. 2011