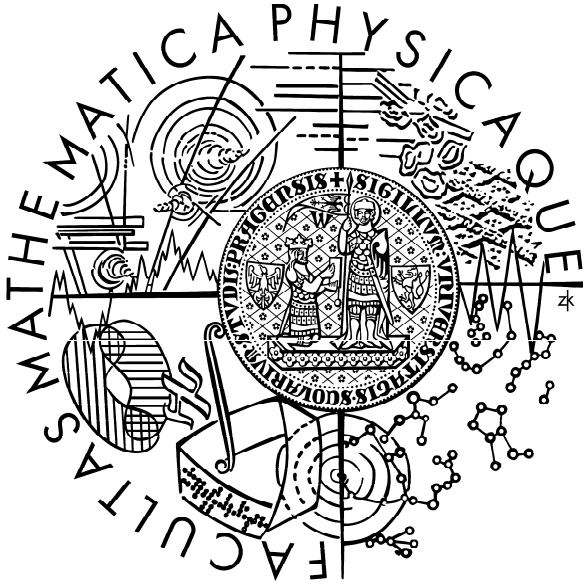
CHARLES UNIVERSITY IN PRAGUE
FACULTY OF MATHEMATICS AND PHYSICS

DOCTORAL DISSERTATION



RNDr. PAVEL SURYNEK

CONSTRAINT PROGRAMMING IN PLANNING

ADVISER: DOC. RNDr. ROMAN BARTÁK, Ph.D.

DEPARTMENT OF THEORETICAL COMPUTER SCIENCE AND
MATHEMATICAL LOGIC

PRAGUE 2008

ACKNOWLEDGEMENT

I would like to thank my supervisor Doc. RNDr. Roman Barták, Ph.D. for his systematic guidance, collaboration, and support. I also would like to thank many anonymous reviewers for providing me a valuable feedback on my research. Finally let me say thank to everyone who supported me during my Ph.D. studies.

I declare that this thesis was composed by myself and all presented results are my own unless otherwise stated.

Pavel Surynek

TABLE OF CONTENTS

Abstract viii

1 Introduction 1
- 1.1 The Thesis in the Context of Artificial Intelligence 2
- 1.2 Contributions 3
- 1.3 Overview by Chapters 4

2 Classical Planning and Constraint Programming 5
- 2.1 Classical Planning 5
  - 2.1.1 Motivating Example 6
- 2.2 Extensions Related to Classical Planning 9
- 2.3 Planning in Practice 10
- 2.4 Representation of Planning Problems 11
  - 2.4.1 Classical Representation 11
  - 2.4.2 Representation Using State Variables 17
- 2.5 Complexity of Planning Problems 20
- 2.6 Overview of Planning Techniques 20
- 2.7 Planning Using Planning Graphs 22
  - 2.7.1 Planning Graphs 23
  - 2.7.2 GraphPlan Algorithm 28
- 2.8 Constraint Programming 33
  - 2.8.1 Constraint Satisfaction Problems 34
  - 2.8.2 Solving Techniques 35
  - 2.8.3 Consistency Techniques 36
  - 2.8.4 Arc-consistency 37
  - 2.8.5 Algorithm AC-3 38
  - 2.8.6 Global Constraints and Problem Modeling 40
  - 2.8.7 Problem Modeling 40

3 Contributions to Planning Using Planning Graphs 42
- 3.1 Problem of Finding Supporting Actions 42
- 3.2 Basic Constraint Model: Arc-consistency 45
  - 3.2.1 Variants of Constraint Propagation 49
  - 3.2.2 Experimental Evaluation 55

3.2.3 Overall Analysis of Results 63
3.2.4 Discussion and Related Works 63
3.3 Advanced Constraint Model: Global Constraints 65
3.3.1 Projection Consistency 65
3.3.2 Preprocessing Step: Clique Decomposition 66
3.3.3 Counting Derived from Clique Decomposition 68
3.3.4 Experimental Evaluation 76
3.3.5 Conclusion and Discussion 78
3.4 Tractable Class of Problem of Finding Supports 78
3.4.1 Tractability 79
3.4.2 Experimental Evaluation 89
3.4.3 Discussion of Results 92
3.5 Difficult Planning Problems 92
3.5.1 Experiments 93
3.6 Summary and Conclusion 96

4 Contributions to Boolean Satisfiability 98
4.1 SAT Reformulation Using Clique Decomposition 100
4.1.1 Inference of Conflicting Literals 101
4.1.2 Clique Decomposition and Literal Contribution Counting 104
4.1.3 Output of the Reformulation Process 106
4.2 Experimental Results 107
4.2.1 Difficult SAT Instances Selected for Experiments 108
4.2.2 Effect of Problem Reformulation 109
4.3 Related Works 111
4.4 Summary and Conclusion 112

5 Conclusions and Future Works 113

Bibliography 115

A Difficult Planning Problems 131

B Removable Medium 136

List of Algorithms

2.1 GRAPHPLAN - PLAN EXTRACTION 29
2.2 GRAPHPLAN 32
2.3 AC-3 38
3.1 SOLVING METHOD FOR BASIC CONSTRAINT MODEL OF PROBLEM OF SUPPORTS 47
3.2 CONSTRAINT PROPAGATION - VARIANT A 50
3.3 CONSTRAINT PROPAGATION - VARIANT B 51
3.4 CONSTRAINT PROPAGATION - VARIANT C 52
3.5 GREEDY CLIQUE COVER ALGORITHM 67
3.6 PROJECTION CONSISTENCY PROPAGATION ALGORITHM 71
3.7 CONSTRAINT PROPAGATION - PROJECTION CONSISTENCY 76
3.8 CONSTRAINT PROPAGATION - STRONG PROJECTION CONSISTENCY 83

List of Figures

2.1 EXAMPLE OF A SIMPLE PLANNING ENVIRONMENT IN WHICH THE PLANNING TASK TAKES PLACE 6
2.2 EXAMPLE OF A GOAL FOR THE SIMPLE PLANNING ENVIRONMENT FROM FIGURE 2.1 7
2.3 EXAMPLE OF A TIME-STEP-OPTIMAL PLAN FOR THE PLANNING PROBLEM 8
2.4 STRUCTURE OF PLANNING GRAPH 27
2.5 ARC-CONSISTENCY OF A BINARY CONSTRAINT 37
3.1 PROBLEM OF FINDING SUPPORTS FOR A SUB-GOAL 43
3.2 REDUCTION OF BOOLEAN SATISFIABILITY TO THE PROBLEM OF FINDING SUPPORTS 45
3.3 COMPARISON OF OVERALL SOLVING TIMES (LOGARITHMIC SCALE) - (STD, VAR-A, VAR-B, VAR-C) 59
3.4 COMPARISON OF PLAN EXTRACTION PHASES TIMES (LOGARITHMIC SCALE) - (STD, VAR-A, VAR-B, VAR-C) 60
3.5 COMPARISON OF NUMBER OF CONSTRAINT CHECKS - (STD, VAR-A, VAR-B, VAR-C) 61
3.6 COMPARISON OF NUMBER OF BACKTRACKS - (STD, VAR-A, VAR-B, VAR-C) 62
3.7 COMPARISON OF IMPROVEMENTS WITH RESPECT TO VARIANT A - (STD, VAR-B, VAR-C) 62
3.8 ILLUSTRATION OF MUTEX GRAPH AND CLIQUE DECOMPOSITION 67
3.9 ILLUSTRATION OF PROJECTION CONSISTENCY 74
3.10 COMPARISON OF OVERALL SOLVING TIMES (LOGARITHMIC SCALE) - (STD, VAR-A, VAR-B, VAR-C, PRJ) 77

3.11 A DIAGRAM OF MERGED POSITIVE EFFECTS OF THE CLIQUE DECOMPOSITION 87
3.12 CLIQUE INTERSECTION GRAPH 88
3.13 COMPARISON OF OVERALL SOLVING TIMES (LOGARITHMIC SCALE) - (STD, VAR-C, PRJ, TRACT) 90
3.14 COMPARISON OF PLAN EXTRACTION PHASE TIMES (LOGARITHMIC SCALE) - (STD, VAR-C, PRJ, TRACT) 90
3.15 COMPARISON OF NUMBER OF BACKTRACKS (LOGARITHMIC SCALE) - (STD, VAR-C, PRJ, TRACT) 91
3.16 COMPARISON OF IMPROVEMENTS WITH RESPECT TO STANDARD GRAPHPLAN - (VAR-C, PRJ, TRACT) 91
4.1 GRAPH OF TRIVIAL CONFLICTS AND INTERMEDIATE GRAPH OF CONFLICTS 103
4.2 FINAL GRAPH OF CONFLICTS 107

List of Tables

2.1 COMPLEXITY OF PLANNING GRAPH 26
2.2 COMPLEXITY OF AC-3 40
3.1 COMPLEXITY OF PROPAGATION IN BASIC CONSTRAINT MODEL 54
3.2 STATISTICAL CHARACTERISTICS COLLECTED DURING EXPERIMENTAL EVALUATION 57
3.3 CHARACTERISTICS OF SOLUTION CONCURRENT PLANS FOR DOCK WORKER ROBOTS PROBLEMS 58
3.4 CHARACTERISTICS OF SOLUTION CONCURRENT PLANS FOR TOWERS OF HANOI PROBLEMS 58
3.5 CHARACTERISTICS OF SOLUTION CONCURRENT PLANS FOR REFUELING PLANES PROBLEMS 59
3.6 PERFORMANCE COMPARISON OF PLANNERS ON DIFFICULT PROBLEMS - PART I 94
3.7 PERFORMANCE COMPARISON OF PLANNERS ON DIFFICULT PROBLEMS - PART II 95
4.1 EXPERIMENTAL COMPARISON OF SAT SOLVERS - PART I 109
4.2 EXPERIMENTAL COMPARISON OF SAT SOLVERS - PART II 110
B.1 CONTENT OF THE ATTACHED REMOVABLE MEDIUM 136

List of Examples

2.1 EXAMPLE OF LANGUAGE, STATE, AND GOAL 12
2.2 EXAMPLE OF ACTIONS 13
2.3 EXAMPLE OF APPLICATION OF AN ACTION 14
2.4 EXAMPLE OF PLANNING OPERATORS AND SUBSTITUTIONS 16
2.5 EXAMPLE OF LANGUAGE, STATE, AND GOAL IN STATE VARIABLE REPRESENTATION 18
2.6 EXAMPLE OF PLANNING OPERATOR IN STATE VARIABLE REPRESENTATION 19
2.7 EXAMPLE OF A CONSTRAINT SATISFACTION PROBLEM 34

ABSTRACT

This thesis deals with planning problems and Boolean satisfiability problems that represent major challenges for artificial intelligence.

Planning problems are stated as finding a sequence of actions that reaches certain goal. One of the most successful techniques for solving planning problems is a concept of *planning graphs* and the related GraphPlan algorithm. In the thesis we identified a weak point of the original GraphPlan algorithm which is the search for actions that support certain goal. We proposed to model this problem as a constraint satisfaction problem and we solve it by maintaining arc-consistency. Several propagation variants for maintaining *arc-consistency* in the model are proposed. The model and its solving process were integrated into the general GraphPlan-based planning algorithm. The performed experimental evaluation showed improvements in order of magnitude in several measured characteristics (overall solving time, number of backtracks) compared to the standard version of the GraphPlan algorithm.

Next we proposed a stronger consistency technique for pruning the search space during solving the problem of finding supports. It is called *projection consistency* and it is based on disentangling the structure of the problem formulation. If the problem of finding supporting actions is interpreted as a graph then this graph is typically well structured - it consists of a small number of relatively large complete sub-graphs. We exploit this structural information to rule out actions from further consideration using a special reasoning. This process reduces the search space significantly. The performed experimental evaluation showed again significant improvements compared to the previous method based on arc-consistency.

Finally, we proposed a special class of the problem of finding supporting actions that can be solved in polynomial time. Thus if the problem satisfies certain conditions it becomes easy to solve (it belongs to the class of tractable problems). We also proposed heuristics that guide the solving process and simplify the problem to become tractable in early stage of the process. Experimental evaluation showed that the method based on preference of this class of problems performs best of all the developed methods. Moreover, some of the planning problems were solved without backtracking. The experiments also showed that this method is competitive with state-of-the-art planners on certain problems.

The contribution of this thesis to solving Boolean satisfiability problems consists in developing a special preprocessing method again based on disentangling the structural information encoded in the problem. The developed method is called *clique consistency* - it is an adaptation of projection consistency for the area of Boolean satisfiability. The preproc-

essing method either decides the problem or passes the simplified problem to the general solving system. The performed experiments showed that the clique consistency technique can improve the solving process of difficult classes of Boolean satisfaction problems significantly in comparison with several state-of-the-art SAT solvers.

CHAPTER 1

INTRODUCTION

This thesis is a presentation of the research results achieved during my Ph.D. studies at Charles University in Prague. The primary topic of this thesis is an application of *constraint programming* in *artificial intelligence planning*. An application of constraint programming techniques to improve solving of planning problems is the main goal of the thesis.

Planning problems represent an intensively studied research area of artificial intelligence. Due to their complexity, which is far beyond tractability, solving planning problems is an area still requiring innovations. Another important motivation for developing more powerful solving techniques for planning problems is their practical importance. The areas of application of planning technology range from industrial planning of manufacturing operations to planning of actions for autonomous planetary exploration agents.

From a wider point view, the research effort was not concentrated on planning problems only but also on improving search for a solution of a certain problem generally. Therefore, results achieved in the area of solving *satisfiability problems* are also presented in the thesis (a satisfiability problem is a task of finding a valuation of variables that satisfies a given logical formula). The satisfiability problems represent a challenge for research in artificial intelligence. They are also used as the main benchmark suit for measuring qualities of improvements in search techniques since large collections of benchmark problems are available. Again, the motivation for improving techniques for solving satisfiability problems is not only theoretical. Real life applications where satisfiability problems must be solved account for example automated software testing or microprocessor verification.

The pivotal concept of both major topics of this thesis (planning problems and satisfiability problems) is *search* for solution. The search for solution can be regarded as an exhaustive testing (systematic or non-systematic) of possible candidates for solution. For today's computational hardware the search seems to be the only option to solve the difficult problems - namely the problems that are *NP*-complete or even more difficult (Cook, 1971). In the thesis we are trying to improve the search for specific problems by specialized techniques. As a result we are able to test more candidates for solution and to solve harder problems.

1.1 The Thesis in the Context of Artificial Intelligence

Let us adopt a simplified intuitive statement that one of the engineering goals of artificial intelligence is to build an intelligent agent that reasons and behaves rationally when it is solving problems (Russell and Norvig, 2003). The rational reasoning and behavior is defined as the reasoning and behavior typical for humans when they are solving problems.

Two types of problems are studied in the thesis - planning problems (Allen *et al.*, 1991; Ghallab *et al.*, 2004) and satisfiability problems (Davis and Putnam, 1960; Cook, 1971). Solving of these problems produces something that can be interpreted as rational reasoning and behavior. For example, acting according to plan which solves a certain planning problem looks like a rational behavior. From this point of view, the topics studied in this thesis contribute to building of such an intelligent agent. And hence contribute to one of the goals of artificial intelligence as a science.

Planning problems are tasks of determining a sequence of actions that reach a certain goal. A planning problem takes place in some kind of an abstraction from the real world. This abstraction represents a simplification of the real world which for example abstracts from the continuous character of the reality - that is typically implemented by the instantaneous character of actions. The actions itself are local transformations of the planning world.

Satisfiability problem is a problem of deciding whether a given Boolean formula has a satisfying valuation of its variables. Thus the solution of such problem is either the answer yes (possibly accompanied with the satisfying valuation) or the answer no.

The main source of inspirations for improvements in solving planning problems and satisfiability problems was another area of artificial intelligence - *constraint programming* (Dechter, 2003). In particular, constraint programming methodology provides a concept of so called *consistency techniques* (or propagation/filtering techniques). Intuitively said, a consistency technique can be used to predict the impact of decisions made during the search on future evolution of the search process. Thus the consistency technique reduces the number of candidates for solution that need to be tested (a series of decisions that does not lead to the solution is identified early).

There are two main approaches how ideas from constraint programming can be applied in another area of problem solving. One approach is to take an existing technique from constraint programming and apply it directly to solve a problem of our choice. Another approach is to regard constraint programming as an inspiration only and develop specialized techniques for specific problems. We used both approached in the thesis. However, the latter approach when we developed a specialized technique inspired by constraint programming proved to be more efficient.

1.2 Contributions

The contribution of this thesis to above topics of artificial intelligence consists in developing of new concepts inspired by constraint programming. The proposed concepts improve the solving process of planning problems and Boolean satisfaction problems in certain cases.

We focused on improving the solving technique for planning problems over so called *planning graphs*. Planning graphs and the related algorithm *GraphPlan* represent one of the most successful concepts for working with so called *concurrent plans* - plans that allow more than one action to be performed in a single time-step. Although the GraphPlan algorithm is considered no longer to be state-of-the-art its building blocks are still used in state-of-the-art planners (for example formulation of problems using planning graphs is still used). We identified some weak points of the GraphPlan algorithm. Namely, the process how supporting actions for a certain goal are searched is very inefficient within the original algorithm (the problem itself is *NP*-complete). We improved this process by formulating the problem as a *constraint satisfaction problem* and solved it using maintaining a certain type of local consistency (*arc-consistency* - Mackworth, 1977).

The relatively successful application of local consistency was an inspiration to develop a more global and more specialized type of consistency for the problem. The new type of consistency is based on disentangling the structure of the problem hidden in its formulation. We visualized the problem as a graph. This uncovered that the graphical structure of the problem of finding supporting actions is far from random. The graph is typically formed by a small number of large complete sub-graphs (plus some additional edges). The consistency that utilizes this structure for solving the problem of supports is described in the thesis. This new technique is called *projection consistency*.

The concept of projection consistency was further improved. We found that it can be used to define a special class of the problem of finding supporting actions that can be solved in polynomial time. A special variant of the solving process for the problem of finding supports within the GraphPlan algorithm was proposed. The solving process utilizes the class of problems solvable in polynomial time. The experimental evaluation unexpectedly showed that this enhancement of GraphPlan algorithm becomes competitive with today's state-of-the-art planners on certain problems (this was unexpected because of a not well optimized testing implementation compared to fine tuned implementations of state-of-the-art planners).

Finally, we applied the experiences gained during experimentation with planning problems in solving Boolean satisfaction problems. We proposed a special preprocessing of satisfiability problems which helps the general solver to decide the problem. It is again based on disentangling the structure of the formulation of the problem - again the problem is interpreted as a graph. Contrary to problems of finding supporting actions the graphical representation of satisfiability problems is unstructured and must be processed further to

make the structures visible. The method for preprocessing satisfiability problems was called *clique consistency* since it is again based on complete sub-graph structures.

All the results presented in the thesis were published in reviewed international conferences CP (Principles and Practice of Constraint Programming), FLAIRS (Florida Artificial Intelligence Research Society International Conference), IICAI (Indian International Conference on Artificial Intelligence), and SARA (Symposium on Abstraction, Reformulation, and Approximation), in the international workshop CSCLP (Annual ERCIM Workshop on Constraint Solving and Constraint Programming), and in the book Recent Advances in Constraints 2007 (selected papers from CSCLP).

1.3 Overview by Chapters

Chapter 2: Classical Planning and Constraint Programming. This chapter is devoted to the general preliminaries for classical planning and constraint programming technology. Formalisms for expressing planning problems and the basic concepts of constraint programming are described in this chapter. ●

Chapter 3: Contributions to Planning Using Planning Graphs. Having the preliminaries from the previous chapter, constraint models for solving the problem of finding supporting actions and related consistency techniques are developed here. Arc-consistency and projection consistency methods are analyzed here theoretically as well as experimentally on the number of planning problems. A special class of the problem of finding supporting actions that can be solved in polynomial time is defined in this chapter. The chapter describes original work. ●

Chapter 4: Contributions to Boolean Satisfiability. This chapter is devoted to studying Boolean satisfaction problems. A special preprocessing method for Boolean satisfaction problems and experimental comparison with state-of-the-art solvers is given in this chapter. The whole chapter describes original work. ●

Chapter 5: Conclusions and Future Work. This chapter is devoted to the summary of the contributions of the thesis and to the discussion the possible future development. ●

Appendix A: Difficult planning problems. Several difficult planning problems that were used for competitive comparison in chapter 3 are described in detail in this appendix. ●

Appendix B: Attached removable medium. This appendix describes the contents of the attached removable medium. ●

CHAPTER 2

CLASSICAL PLANNING AND CONSTRAINT PROGRAMMING

This chapter is devoted to the basic preliminaries for *classical planning* and *constraint programming* technology. This chapter should be regarded as the introduction to existing concepts upon which the main parts of the thesis builds.

2.1 Classical Planning

Planning problems (Allen *et al.*, 1991; Ghallab *et al.*, 2004) rank among the most challenging classes of problems arising in artificial intelligence. The planning problem is stated as a task of determining a sequence of actions for a certain agent (or a group of agents) whose execution achieves a given goal from a given initial state. This sequence of actions is called a *plan* in this context. For simplicity, it is assumed that the planning task takes place in a *fully observable*, *static* and *deterministic* environment that represents an abstraction from the real world. A fully observable environment is such that a reasoning mechanism has the complete information about the state of the environment. A static environment is such an environment that can be changed only by the activity of an agent we plan for. In other words, the environment is not changed by any out of control external activities. Finally, a deterministic environment is such an environment in which the results of the agent's activity can be anticipated completely.

Even using such a level of abstraction the planning problems remain computationally extremely difficult. The task of finding a plan is generally intractable (*EXPSPACE*-complete) or even undecidable depending on the expressivity of tools for describing the problems (Ghallab *et al.*, 2004; Erol *et al.*, 1996). This theoretical difficulty of planning problems may evoke that we are unable to solve these problems automatically using computers of today's architecture. However, even relatively large complex planning problems can be

solved in practice by today's automated solvers. This success is achieved by incorporating efficient and intelligent algorithmic techniques into the automated solver.

By solving planning problems we typically mean the task of finding optimal plans. The step-optimal planning produces the shortest possible plans that reach the given goal.

2.1.1 Motivating Example

For a quick introduction into planning problems, consider an example depicted in figure 2.1 where we have two locations equipped with cranes and connected by a road. Next, there is a truck and several boxes stacked in piles. The cranes can manipulate boxes arbitrarily using their manipulating hand and the truck can move between locations possibly carrying some boxes (at most two). The description of the state of the planning environment must comprise what boxes are at what location, which box is laid on top of another box, whether a crane is holding some box etc. That is, we need to know what are the relations among the objects and agents in the environment.

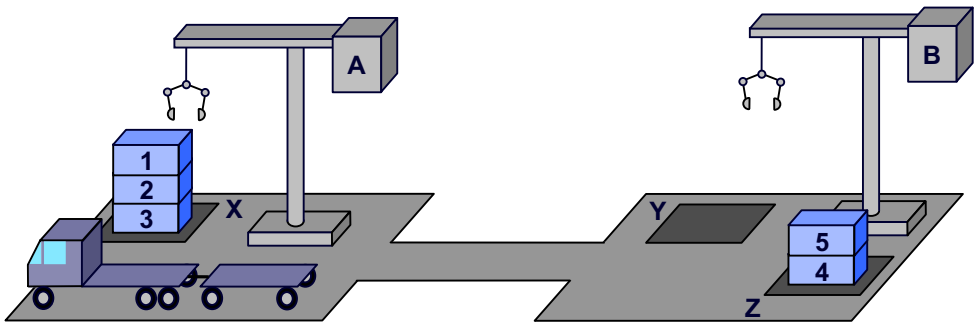


Figure 2.1. *EXAMPLE OF A SIMPLE PLANNING ENVIRONMENT IN WHICH THE PLANNING TASK TAKES PLACE.* The environment consists of two locations with several boxes stacked in piles. The locations are connected by a road. Each location is equipped with a crane. There is a truck which can carry at most two boxes - one box on itself and one on its trailer. The cranes can manipulate boxes (possible actions are for instance *take a box, put a box*) and the truck can carry boxes from one location to another location (the corresponding action is *move a truck*). The actions are instantaneous, that is, we abstract from continuous execution of an action. Furthermore, we abstract from other properties such as exact positions of the objects in the environment.

On the other hand, it is possible to abstract from exact positions of boxes within the location without affecting the ability to reason about the problem with respect to the given goal. The similar abstraction can be done in regard of actions. Since only the information about sequencing of *take a box* (with specifying what a box) and *put the box* (with specifying on what a box) actions is relevant when reasoning about the goal, it is possible to ab-

stract from exact controlling of the robot motion as a continuous event. An action is an instantaneous procedure. That is when an action is applied we immediately obtain the results.

The whole planning environment is fully observable for a reasoning procedure. That is, the reasoning procedure sees the whole picture of the environment as for example shown in figure 2.1 (knows what relations among objects and agents are currently satisfied). The only change of the environment can be made through acting of agents (cranes and truck in the figure 2.1 are the agents that can change the environment). The agents act through allowed actions. The result of agent's actions can be fully anticipated (the result of a *move* action is always the movement of the truck to the specified location).

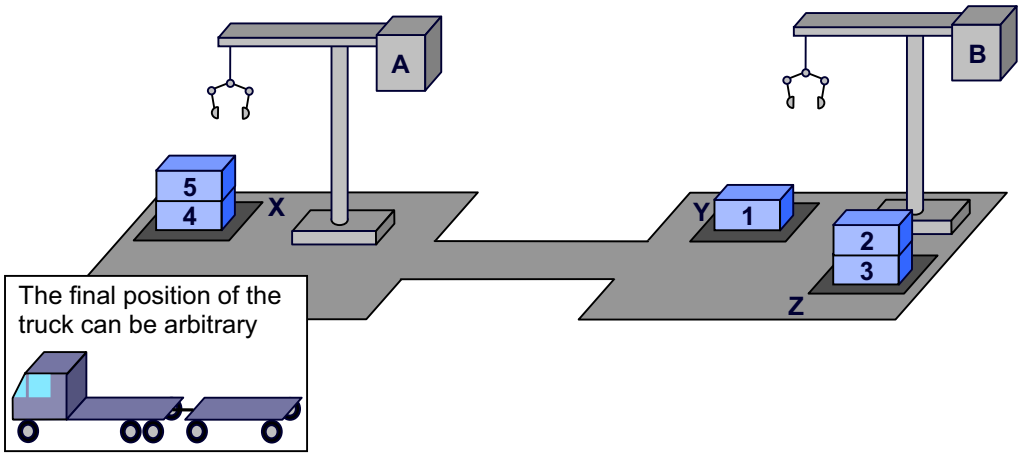
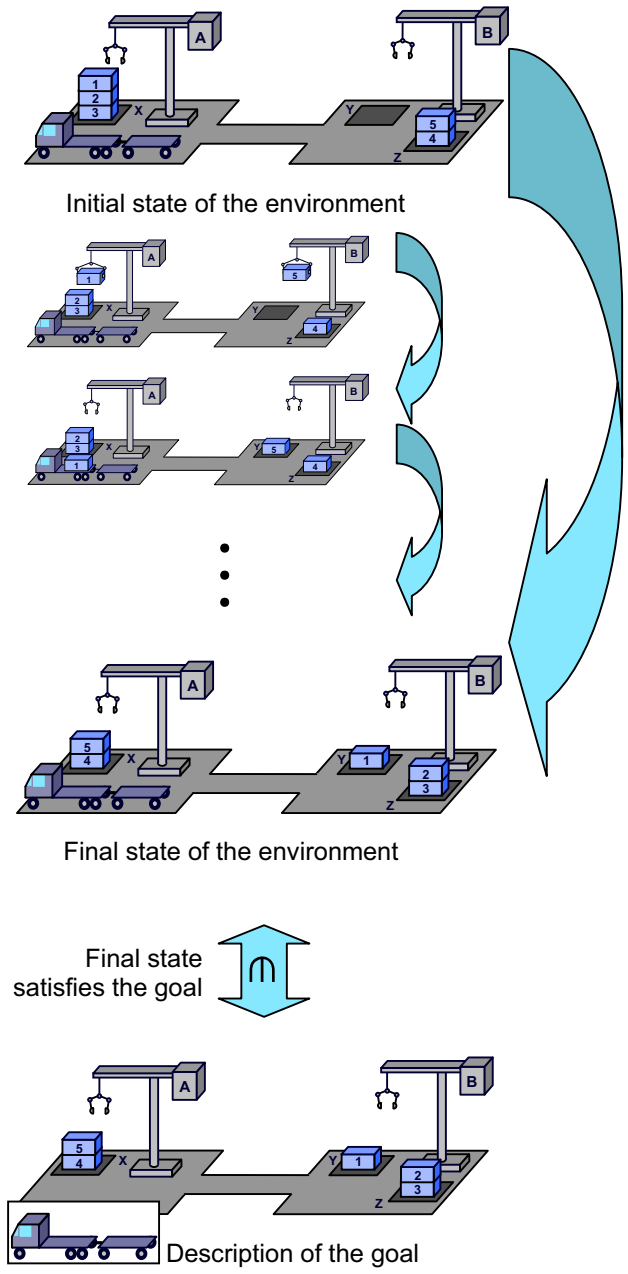


Figure 2.2. *EXAMPLE OF A GOAL FOR THE SIMPLE PLANNING ENVIRONMENT FROM FIGURE 2.1.* The boxes originally stacked in a single pile at the right location are required to be split in two piles at the right location. The similar condition is required for boxes originally stacked at the right location; these are required to be stacked at the left location. The depicted goal does not care about the final position of the truck.

The planning problem is a task of finding a sequence of actions that transforms the given initial state of the planning environment into the state satisfying a *goal*. The goal can be regarded as a description of a set of states of the environment that we consider as satisfactory for meeting our intention. See for instance figure 2.2 where the goal state of the planning environment with cranes and truck is depicted. The goal shown in the figure requires that boxes originally stacked in a single pile at the left location are now stacked in two piles at the right location. The similar condition is required for the boxes originally stacked at the right location; these are required to be at the left location. Nevertheless, the goal specifies nothing about the final position of the truck since we do not care about that. In other words, every state of the planning environment with boxes positioned according to the requirements of the goal and with arbitrary position of the truck satisfies the goal.



Plan for reaching the goal

1. take(crane_A, box_1, pile_X)
   take(crane_B, box_5, pile_Z)
2. load(truck, crane_A, box_1)
   put(crane_B, box_5, pile_Y)
3. move(truck, site_A, site_B)
   take(crane_B, box_4, pile_Z)
   take(crane_A, box_2, pile_X)
4. load(truck, crane_B, box_4)
   put(crane_A, box_2, pile_X)
5. take(crane_B, box_5, pile_Y)
   take(crane_A, box_2, pile_X)
6. put(crane_B, box_5, pile_Z)
7. put(crane_A, box_2, pile_X)
   unload(truck, crane_B, box_1)
8. move(truck, site_B, site_A)
   put(crane_B, box_1, pile_Y)
   take(crane_A, box_2, pile_X)
9. load(truck, crane_A, box_2)
   take(crane_B, box_1, pile_Y)
10. move(truck, site_A, site_B)
    put(crane_B, box_1, pile_Y)
    take(crane_A, box_3, pile_X)
11. put(crane_A, box_3, pile_X)
    unload(truck, crane_B, box_2)
12. move(truck, site_B, site_A)
    put(crane_B, box_2, pile_Y)
    take(crane_A, box_3, pile_X)
13. load(truck, crane_A, box_3)
    take(crane_B, box_5, pile_Z)
14. put(crane_B, box_5, pile_Y)
    unload(truck, crane_A, box_4)
15. move(truck, site_A, site_B)
    take(crane_B, box_5, pile_Y)
16. load(truck, crane_B, box_5)
    put(crane_A, box_4, pile_X)
17. take(crane_A, box_4, pile_X)
    unload(truck, crane_B, box_3)
18. move(truck, site_B, site_A)
    put(crane_A, box_4, pile_X)
    put(crane_B, box_3, pile_Z)
19. take(crane_B, box_2, pile_Y)
    unload(truck, crane_A, box_5)
20. put(crane_A, box_5, pile_X)
    put(crane_B, box_2, pile_Z)

Figure 2.3. *EXAMPLE OF A TIME-STEP OPTIMAL PLAN FOR THE PLANNING PROBLEM.* The plan solves the planning problem with the initial state from figure 2.1 and with the goal from figure 2.2. When an action is executed in a state of the environment, the result is another state changed accordingly. The problem is solved if we obtain a sequence of actions that reaches some state satisfying the given goal. The plan shown here is step-optimal (20 steps are necessary). That is, no shorter plan in terms of steps exists.

Altogether, the planning problem consists of three components. First, we are given an initial state of the planning environment (as for example shown in figure 2.1). Second, we are given a goal (as shown in figure 2.2). Finally, we have a set of allowed actions through which the planning environment can be changed (for example *take a box*, *put the box*, etc.). The task is to find a sequence of actions from the given set that transforms the given initial state of the planning environment into a state that meets the requirements of the given goal. The sequence of actions that achieves the goal is called a plan. An example of the plan is shown in figure 2.3.

When an action is applied on a state of the planning environment, the result is another state changed accordingly. Similarly it is possible to apply a sequence of actions on a state, which is done by applying actions one by one on the current state provided we start with the original state. It is also possible to apply a set of non-interfering actions to a state simultaneously (actions that do not influence each other's results). The evolution of the initial state of the environment through several middle states to the final state is shown in figure 2.3. At each step a set of actions can be applied simultaneously (or in arbitrary order one by one). If the resulting final state satisfies the goal, the corresponding sequence of actions is solution - a plan for the given problem. Moreover, the plan shown in figure 2.3 is step-optimal (exactly 20 steps are necessary to solve the problem). That is, no plan of fewer steps exists.

To appreciate the difficulty of the task of finding a step-optimal plan the reader may try to find another step-optimal plan or to prove that the plan shown in the figure is really a step-optimal plan.

The problem and its solution we have just intuitively introduced are often called *classical planning* (Ghallab *et al.*, 2004). By this we mean a problem in which it is reasoned about instantaneous actions and their sequencing in a deterministic environment only. We do not care about durations of the actions, uncertainty and so on in this classical approach. However, there exist several extended concepts of planning where the aspects such as time play an important role.

2.2 Extensions Related to Classical Planning

There were several attempts to make the reasoning about plans more realistic and more effective for particular situations. Let us briefly summarize them in this section.

Temporal planning represents an extension of classical planning which tries to handle time more realistically and more effectively. Contrary to instantaneous character of actions in classical planning, actions are considered to have durations in temporal planning. The most influential concepts from this area are presented in (Vilain and Kautz, 1986), (Allen, 1983), (Dechter *et al.*, 1989), and (El-Kholy and Richards, 1996; Liatsos and Richards, 1999).

Planning with resources (Bacchus and Ady, 2001) is another technique for capturing properties of the real planning world. Planning with resources is trying to effectively handle sets of entities appearing in the planning environment which behave equally and which can be consumed or borrowed (Ghallab *et al.*, 2004). A *resource* can be for example *fuel*, a *set of transporters* or *machines* etc. Equality of resources means that resources of the same type are interchangeable.

Another paradigm of planning incorporates *uncertainty* into the reasoning about plans. This is motivated by practice since in practice not all planning environments are fully observable or fully deterministic (Kaelbling *et al.*, 1998). Uncertainty in planning is often modeled using *Markov decision processes* (Kaelbling *et al.*, 1995).

Recently so called *probabilistic planning* is becoming widely studied (as it is evidenced by the Probabilistic planning track in the *IPC - International Planning Competition* (Gerevini *et al.*, 2006)). An effect of the action on the planning environment is a random variable of a given distribution in the probabilistic planning (Blum and Langford, 2000; Little *et al.*, 2005; Little and Thiébaux, 2006).

Hierarchical task network planning uses a different view of the problem (Erol *et al.*, 1994a, 1994b, 1996; Ghallab *et al.*, 2004). The problem of hierarchical planning consists in finding decompositions of compound tasks into simpler tasks which eventually leads to a sequence of actions. This conceptual approach is often used in practice.

2.3 Planning in Practice

Planning techniques are necessary when autonomous behavior of some agent is necessary. The most prominent example of this kind is distance space and planetary exploration. Issues concerning planning for space exploration are studied in (Jónsson *et al.*, 2000; Frank *et al.*, 2001; Frank and Jónsson, 2003).

Particularly, *Deep Space 1* was a project where advanced planning techniques were extensively used. The *Deep Space 1* was a space exploration device designed to observe a Borrelli comet. It was equipped with an autonomous system for making decisions regarding navigation, exploration, monitoring, fault-diagnosis, and re-planning (Nayak *et al.*, 1998; Muscettola *et al.*, 1998).

Another example of successful space exploration project that used advanced planning technology was the Mars planet exploration by *Spirit* and *Opportunity* rovers (Ai-Chang *et al.*, 2004). *Spirit* and *Opportunity* were six-wheeled rovers designed to make surface explorations on the planet *Mars*.

Both examples from space exploration share a common attribute that a complete autonomous behavior of the agent is required since human remote control is not applicable because of the distance.

The complete autonomous behavior of an agent that does not need human control is also a goal for military specialists. Currently unmanned combat air vehicle and other devices are under development (Boeing, 2003a, 2003b).

Space exploration and military operations are not the only areas where advanced planning techniques are used or are intended to be used. Recently a successful competition DARPA Grand Challenge (DARPA, 2007a, 2007b) showed that artificial intelligence can be used to drive cars autonomously (Thrun *et al.*, 2006).

Planning techniques are also successfully used in industry for solving and optimizing difficult manufacturing or logistic processes (Nau *et al.*, 1995; Muñoz-Avila *et al.*, 2001).

A more detailed survey of artificial intelligence techniques for practical applications is given in (Kumar, 2005).

2.4 Representation of Planning Problems

A formal description of planning problems in classical representation is given in this section. Two representations of planning problems are introduced - a *classical representation* and a *representation using state variables* (Ghallab *et al.*, 2004).

2.4.1 Classical Representation

To describe a planning problem in a certain planning environment we use a language L with finitely many predicate, variable, and constant symbols in classical representation. The language L is associated with the planning environment and it is possibly different for different environments. The finite set of predicate symbols of the language L is denoted as P_L, the finite set of symbols for variables is denoted as V_L, and the finite set of constants is denoted as C_L.

Constants from the set C_L represent objects appearing in the planning environment. Predicate symbols from the set P_L are used to express relations among objects. The variable symbols are auxiliary constructs. They are used in construction of so called *planning operators* which are some kind of a generic action. The language has no function symbols. The following definitions assume a fixed language L (that is, we are in a fixed planning environment).

Definition 2.1 (TERM, ATOM, AND LITERAL). A *term* t is either a variable symbol or a constant symbol (that is $t \in C_L \cup V_L$). An *atomic formula* is a construct of the form $p(t_1,t_2,\ldots,t_n)$, where p is a predicate symbol ($p \in P_L$) and t_i is a term for every $i = 1,2,\ldots,n$; the number n is called an *arity* of the predicate symbol. Atomic formulas are called *atoms* in short. A *literal* is an atom or the negation of an atom. □

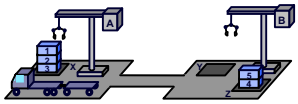
Definition 2.2 (GROUND TERM, ATOM, AND LITERAL). A term t is *ground* if it is a constant symbol (that is $t \in C_L$). An *atom* $p(t_1, t_2, \ldots, t_n)$ is *ground* if t_i is a ground term for every $i = 1, 2, \ldots, n$. A *ground literal* is a ground atom or the negation of a ground atom. □

Definition 2.3 (STATE, GOAL, AND GOAL SATISFACTION). A *state* is a finite set of ground atoms. A *goal* is a finite set of ground literals. Let g be a goal then g^+ denotes a set of positive literals of g and g^- denotes a set of negative literals of g. The goal g is *satisfied* in the state s if $g^+ \subseteq s \wedge g^- \cap s = \emptyset$. □

States provide a formal description of a situation in the planning environment - it is a snapshot of the planning environment at a certain moment. The goal is a formal description of a situation of the planning environment which we want to establish. There are no variables in states and goals (all these constructs must be ground).

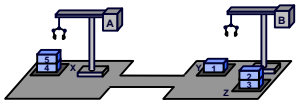
An example of a language, states and goals for description of a planning environment from figure 2.1 is shown in the following example.

Example 2.1. *EXAMPLE OF LANGUAGE, STATE, AND GOAL.* The planning environment is taken from figures 2.1 and 2.2. Predicate symbols are listed together with their arities. In the goal there is no negative literal since it is not necessary for the goal specified by the figure. Notice again that we do not care about the final position of the truck.



A state

C_L=*{truck, locationA, locationB, craneA, craneB, stackX, stackY, stackZ, box1, box2, box3, box4, box5, zero, one, two, nothing}*
P_L=*{at/2, on/2, onTop/2, atBottom/2, loaded/2, holding/2, capacity/2, reachable/2}*



A goal

Example of a state: *{at(truck, locationA), capacity(truck, two), atBottom(box3, stackX), atBottom(box4, stackZ), atBottom(nothing, stackY), on(box1, box2), on(box2, box3), on(box5, box4), onTop(box1, stackX), onTop(box5, stackZ), holding(craneA, nothing), holding(craneB, nothing), reachable(stackX, craneA), reachable(stackY, craneB), reachable(stackZ, craneB)}.*

Example of a goal: *{atBottom(box1, stackY), atBottom(box4, stackX), atBottom(box3, stackZ), on(box5, box4), on(box2, box3), onTop(box1, stackY), onTop(box2, stackZ), onTop(box5, stackX), holding(craneA, nothing), holding(craneB, nothing)}*

The states of the planning environment are changed by actions. Actions formally define possible transitions between the states. The description of an action comprises a condition that must be satisfied before an action can be applied and the eventual change of the

planning environment if the action can be applied. An action applied to a state results into a new state.

Definition 2.4 (ACTION, APPLICABILITY, AND ACTION APPLICATION). An *action* a is a triple $(p(a), e^+(a), e^-(a))$, where $p(a)$ is called a *precondition* of the action, $e^+(a)$ is a *positive effect* of the action, and $e^-(a)$ is a *negative effect* of the action. All the three components of an action are finite sets of ground atoms. An action a is *applicable* to the state s if $p(a) \subseteq s$. The *result of the application* of the applicable action a to the state s is a new state that will be denoted as $\alpha(s,a)$, where $\alpha(s,a) = (s - e(a)^-) \cup e(a)^+$. □

The condition when an action can be applied to a state is represented by its precondition. The eventual change of the state is described by action's positive and negative effects. Positive effects are the ground atoms added to the state and negative effects are the ground atoms deleted from the state. It is required that $e(a)^- \cap e(a)^+ = \emptyset$ for any action a.

The following example shows actions for the planning environment from figure 2.1.

Example 2.2. *EXAMPLE OF ACTIONS.* Actions that take a box by a crane and load the truck with a box using a crane is described below as a triple consisting of precondition, positive effects, and negative effects.

---

*An action for **taking box1 on top of stackX by craneA**.*
(p(take_box1_stackX_craneA), e^+(take_box1_stackX_craneA),
e^-(take_box1_stackX_craneA))=
(
{reachable(stackX, craneA), holding(craneA, nothing), onTop(box1, stackX), on(box1, box2)};
{holding(craneA, box1), onTop(box2, stackX)};
{holding(craneA, nothing), onTop(box1, stackX), on(box1,box2)}
)

*An action for **loading the empty truck with box1 by craneA at locationA**.*
(p(load_truck_box1_craneA_locationA), e^+(load_truck_box1_craneA_locationA),
e^-(load_truck_box1_craneA_locationA))=
(
{holding(craneA, box1), at(truck, locationA), capacity(truck, two)};
{loaded(truck, box1), capacity(truck, one), holding(craneA, nothing)};
{holding(craneA, box1), capacity(truck, two)}
)

The next example shows the process of application of the action to a state. Again the environment from figure 2.1 is used in the example.

Example 2.3. *EXAMPLE OF APPLICATION OF AN ACTION.* A process of application of an action is shown with action *take* from the example 2.2. First the applicability is checked. Then a new state is produced.

*The action **take_box1_stackX_craneA** from the example 2.2 is applicable to the state **s** from the example 2.1 since*

***p(take_box1_stackX_craneA)**={reachable(stackX, craneA), holding(craneA, nothing), onTop(box1, stackX), on(box1, box2)}*

is a proper subset of

***s** = {at(truck, locationA), capacity(truck, two), atBottom(box3, stackX), atBottom(box4, stackZ), atBottom(nothing, stackY), **on(box1, box2)**, on(box2, box3), on(box5, box4), **onTop(box1, stackX)**, onTop(box5, stackZ), **holding(craneA, nothing)**, holding(craneB, nothing), **reachable(stackX, craneA)**, reachable(stackY, craneB), reachable(stackZ, craneB)}.*

*The result of the application of the **take_box1_stackX_craneA** action on the state **s** is the following new state **q***

***q** = {at(truck, locationA), capacity(truck, two), atBottom(box3, stackX), atBottom(box4, stackZ), atBottom(nothing, stackY), ~~on(box1, box2)~~, on(box2, box3), on(box5, box4), ~~onTop(box1, stackX)~~, onTop(box5, stackZ), ~~holding(craneA, nothing)~~, holding(craneB, nothing), **holding(craneA, box1), onTop(box2, stackX)**, reachable(stackX, craneA), reachable(stackY, craneB), reachable(stackZ, craneB)}.*

In classical planning the set of actions is described using a construct called a planning operator. The planning operator represents some kind of a parameterized action. The reason for having operators can be seen again in the environment from example 2.1. We need an action for every combination of the objects for which the action is relevant (for example the operation which *takes a box by a crane* needs to be represented by a *take action* for every pair of a *box* and a *crane*). To define planning operators we need an auxiliary definition of *substitution* of variables for terms.

Definition 2.5 (SUBSTITUTION OF VARIABLES BY TERMS). A *substitution* of variables by terms θ is a partial function assigning terms to the variables. That is $\theta : V_L \to C_L \cup V_L$. A substitution is often described using enumeration of assignments for individual variables in the form $\theta = \{v_1/t_1, v_2/t_2, \ldots, v_n/t_n\}$ where $v_i \in V_L \wedge t_i \in V_L \cup C_L$ for every $i = 1,2,\ldots,n$ and $v_i \neq v_j$ for every $i, j = 1,2,\ldots,n \wedge i \neq j$. The substitution θ is called a *ground substitution* if it substitutes variables for constants, that is $t_i \in C_L$ for every $i = 1,2,\ldots,n$. □

The following notation is usually used for substitutions. For a variable $u \in V_L$ and a substitution θ the notation $u\theta$ stands for $\theta(u) = t_i$ if $u = v_i$ for some $i \in \{1,2,\ldots,n\}$ otherwise $\theta(u) = u$ (the substitution identifier written at the end of a construct denotes the application of the substitution on the construct). For a constant $c \in C_L$ and a substation θ the notation $c\theta$ stands for $\theta(c) = c$. The application of the substitution on atoms is defined as follows. Let $p(t_1,t_2,\ldots,t_n)$ be an atom then $p(t_1,t_2,\ldots,t_n)\theta$ is defined as $p(t_1\theta,t_2\theta,\ldots,t_n\theta)$. Similarly if we have a finite set $\{p_1, p_2,\ldots, p_m\}$ where p_i is an atom for every $i = 1,2,\ldots,n$ then $\{p_1, p_2,\ldots, p_m\}\theta$ is defined as $\{p_1\theta, p_2\theta,\ldots, p_m\theta\}$.

Having the notion of substitutions of variables for terms we are ready to introduce planning operators.

Definition 2.6 (OPERATOR, OPERATOR APPLICABILITY, OPERATOR APPLICATION). A planning operator o is a triple $(p(o), e^+(o), e^-(o))$, where $p(o)$ is called a *precondition* of the planning operator, $e^+(o)$ is a *positive effect* of the planning operator, and $e^-(o)$ is a *negative effect* of the planning operator. All the three components of an operator are finite sets of atoms (contrary to action we allow non-ground atoms here). An operator o is *applicable* to the state s if there exists a substitution θ from variables to ground terms (constants) such that $(p(o), e^+(o), e^-(o))\theta$ is an action (that is all the three components $p(o)\theta$, $e^+(o)\theta$, and $e^-(o)\theta$ are sets of ground atoms) and $p(o)\theta \subseteq s$. The *result of the application* of the applicable operator o with the substitution θ to the state s is a new state that will be denoted as $\alpha(s,o,\theta)$, where $\alpha(s,o,\theta) = (s - e(o)^-\theta) \cup e(o)^+\theta$. □

Planning operators are used for more compact representation of actions since through substitutions it is possible to capture many combinations of objects for which the operation is relevant.

Example 2.4 shows planning operators and corresponding substitutions for the planning environment from figure 2.1.

There are also assumed so called *no-op actions* which represent no operations. Their usage is explained in the context of planning graphs. For every ground atom t we assume a no-op action $noop_t = (t,t,\emptyset)$. That is, a no-op action preserves the given ground atom.

All the ingredients are now ready to define the *planning problem* formally. The planning problem is a task of transforming a given initial state into a state satisfying the goal supposed that only the planning operators from the set of allowed planning operators are used (the set of allowed planning operators determines the set of allowed actions).

Definition 2.7 (PLANNING PROBLEM). A *planning problem* P is a triple (s_0, g, O), where s_0 is an *initial state*, g is a goal and O is a finite set of allowed planning operators. □

The *solution* of a planning problem is a sequence of actions (ground operators) such that when they are sequentially applied starting in the initial state the state resulting at the end satisfies the given goal. The formal definition is as follows.

Example 2.4. *EXAMPLE OF PLANNING OPERATORS AND SUBSTITUTIONS.* The operators represent parameterized version of actions from example 2.2. Variables are written using capitals. The substitutions applied on the operators results into actions from example 2.2.

A planning operator for ***taking a box on top of a stack by a crane****.*
(p(take), e^+(take), e^-(take))=
(
{reachable(STACK, CRANE), holding(CRANE, nothing), onTop(BOX1, STACK), on(BOX1, BOX2)};
{holding(CRANE, BOX1), onTop(BOX2, STACK)};
{holding(CRANE, nothing), onTop(BOX1, STACK), on(BOX1,BOX2)}
)

Substitution $\boldsymbol{\theta_1}$*={CRANE/craneA, BOX1/box1, BOX2/box2, STACK/stackX}*
take/*$\boldsymbol{\theta_1}$ *= take_box1_stackX_craneA

An action for ***loading an empty truck with a box by a crane at a location****.*
(p(load), e^+(load), e^-(load))=
(
{holding(CRANE, BOX), at(TRUCK, LOCATION), capacity(TRUCK, two)};
{loaded(TRUCK, BOX), capacity(TRUCK, one), holding(CRANE, nothing)};
{holding(CRANE, BOX), capacity(TRUCK, two)}
)

Substitution $\boldsymbol{\theta_2}$*={CRANE/craneA, BOX/box1, LOCATION/locationA, TRUCK/truck}*
take/*$\boldsymbol{\theta_2}$ *= load_truck_box1_craneA_locationA

Definition 2.8 (APPLICATION OF A SEQUENCE OF ACTIONS, SOLUTION). We inductively define *application of a sequence of actions* $\phi = (a_1, a_2, \ldots, a_n)$ to a state s_0 in the following way: a_1 must be applicable to s_0, let us inductively denote the result of application of the action a_i to the state s_{i-1} as s_i for all $i = 1,2,\ldots,n$; the condition that a_i is applicable to the state s_{i-1} for all $i = 2,3,\ldots,n$ must hold. The *result of application of the sequence of actions* ϕ to the state s_0 is the state s_n. Sequence $\xi = (a_1, a_2, \ldots, a_n)$ is a *solution* of the planning problem $P = (s_0, g, O)$ if the sequence ξ is applicable to the initial state s_0 and the goal g is satisfied in the result of application of the sequence ξ and a_i is a ground instance of some operator from O for every $i = 1,2,\ldots,n$. □

2.4.2 Representation Using State Variables

There exists another important representation of classical planning problems which is based on the functional view of the planning environment and planning operators (Ghallab *et al.*, 2004). Consider for instance the atom *at(TRUCK, LOCATION)* representing position of a truck in the planning environment from figure 2.1. It never happens (supposing that the set of planning operators is correctly formed) that more than one *at(TRUCK, LOCATION)* atoms occur in a state for a single truck (truck can never be at more than one location at once). That is, the relation *at(TRUCK, LOCATION)* is a representation a function that assigns just one location to a single truck at a moment. This property is very common for the objects from the real world. The position of the truck is a *state variable* in this context.

The language S of the state variable representation consists of finitely many symbols for state variables denoted as F_S, finitely many symbols for constants denoted as C_S, and finitely many symbols for variables denoted as V_S. Again the set of constants C_S represents a set of objects appearing in the planning environment. Variables from the set V_S are used for expressing planning operators in the state variable representation. There are no function symbols in the language of state variable representation. Elements of $V_S \cup C_S$ are called terms. The following definitions suppose a fixed language S.

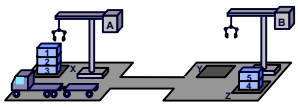
Definition 2.9 (STATE VARIABLE). A *state variable* is a construct of the form $f(t_1,t_2,\ldots,t_n)$ where $f \in F_S$ is a function variable symbol and t_i is a term for every $i = 1,2,\ldots,n$; the number n determines the *arity* of the state variable. If every t_i for $i = 1,2,\ldots,n$ is a ground term then $f(t_1,t_2,\ldots,t_n)$ is a *ground state variable*. Each ground state variable $f(t_1,t_2,\ldots,t_n)$ has assigned a *domain* $D_{f(t_1,t_2,\ldots,t_n)} \subseteq C_S$ of values. □

Definition 2.10 (STATE, GOAL, AND GOAL SATISFACTION IN STATE VARIABLE REPRESENTATION). Let X be a set of all ground state variables over the language S. A *state* in the state variable representation is a set of assignments $\{x = d_x \mid x \in X\}$ where $(\forall x \in X)\, d_x \in D_x$. Let Y be a subset of state variables (we do not require the state variables to be ground). A *goal* in state variable representation is a set of assignments $\{y = d_y \mid y \in Y\}$ where $(\forall y \in Y)\, d_y \in C_S$. The goal g *is satisfied* in the state g if there exists a substitution θ of variables for terms such that $g\theta \subseteq s$ (where the application of a substitution θ on the set of assignments is defined as follows $\{y = d_y \mid y \in Y\}\theta = \{y\theta = d_y \mid y \in Y\}$ where $(\forall y \in Y)\, d_y \in D_{y\theta}$). □

The meaning of states and goals is the same as in the classical representation. The state represents a snapshot of the planning environment at a certain moment while the goal describes a condition on a state we want to achieve.

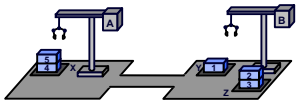
The following example shows language, state and goal in state variable representation for the planning environment from figure 2.1.

Example 2.5. *EXAMPLE OF LANGUAGE, STATE, AND GOAL IN STATE VARIABLE REPRESENTATION.* The planning environment is taken from figures 2.1 and 2.2. State variables are listed together with their arities. Notice that a box can be loaded either in a *truck* or *elsewere* (that is in some of the stacks).



A state

C_S=*{truck, locationA, locationB, craneA, craneB, stackX, stackY, stackZ, box1, box2, box3, box4, box5, zero, one, two, nothing, elsewhere}*
F_S=*{at/1, on/1, onTop/1, atBottom/1, loaded/1, holding/1, capacity/1, reachable/2}*



A goal

Example of a state: *{at(truck) = locationA, capacity(truck) = two, atBottom(stackX) = box3, atBottom(stackZ) = box4, atBottom(stackY) = nothing, on(box2) = box1, on(box3) = box2, on(box4) = box5, onTop(stackX) = box1, onTop(stackZ) = box5, holding(craneA) = nothing, holding(craneB) = nothing, loaded(box1) = elsewhere, loaded(box2) = elsewhere, loaded(box3) = elsewhere, loaded(box4) = elsewhere, loaded(box4) = elsewhere, reachable(stackX, craneA) = true, reachable(stackY, craneB) = true, reachable(stackZ, craneB) = true}.*

Example of a goal: *{atBottom(stackY) = box1, atBottom(stackX) = box4, atBottom(stackZ) = box3, on(box4) = box5, on(box3) = box2, onTop(stackY) = box1, onTop(stackZ) = box2, onTop(stackX) = box5, holding(craneA) = nothing, holding(craneB) = nothing}*

As in the classical representation the states of the planning environment are changed by actions. The following definition formalizes actions in state variable representation.

Definition 2.11 (ACTION, APPLICABILITY, AND ACTION APPLICATION IN STATE VARIABLE REPRESENTATION). An *action* a in state variable representation is a pair $(p(a), e(a))$, where $p(a)$ is a *precondition* of the action, $e(a)$ is an *effect* of the action. The first component $p(a)$ is a finite set of assignments $\{z = d_z \mid z \in Z\}$, where Z is a subset of ground state variables and $(\forall z \in Z)\, d_z \in D_z$. The second component $e(a)$ is a finite set of state transitions $\{w \leftarrow d_w \mid w \in W\}$ where W is a subset of ground state variables and $(\forall w \in W)\, d_w \in D_w$. An action a is *applicable* to the state s if $p(a) \subseteq s$. The *result of the application* of the applicable action a to the state s is a new state that will be denoted as $\chi(s,a)$, where $\chi(s,a) = (s - \{w = c \mid w \leftarrow d \in e(a) \wedge c \in D_x\}) \cup \{w = d \mid w \leftarrow d \in e(a)\}$ (that is, the assignments listed in the effect of the action are updated). □

Definition 2.12 (OPERATOR, OPERATOR APPLICABILITY, AND OPERATOR APPLICATION IN STATE VARIABLE REPRESENTATION). A *planning operator* o in the state variable representation is a pair $(p(a), e(a))$, where $p(a)$ is a *precondition* and $e(a)$ is an *effect* of the

planning operator. The first component $p(a)$ is a finite set of assignments $\{z = d_z \mid z \in Z\}$ where Z is a subset of state variables (we do not require the state variables to be ground here) and $(\forall z \in Z)\, d_z \in D_z$. The second component $e(a)$ is a finite set of state transitions $\{w \leftarrow d_w \mid w \in W\}$ where W is a subset of state variables (again they are not necessarily ground) and $(\forall w \in W)\, d_w \in D_w$. A planning operator o is *applicable* to the state s if there exists a substitution θ from the set of variables V_S to the set of constants C_S such that $p(a)\theta \subseteq s$. The *result of the application* of the applicable planning operator o with the substitution θ to the state s is a new state that will be denoted as $\chi(s,o,\theta)$, where $\chi(s,o,\theta) = (s - \{w = c \mid w \leftarrow d \in e(a)\theta \wedge c \in D_x\}) \cup \{w = d \mid w \leftarrow d \in e(a)\theta\}$ (that is, the assignments listed in the effect of the operator are updated). □

The example below shows planning operators for the planning environment from figure 2.1 in the state variable representation.

Example 2.6. *EXAMPLE OF PLANNING OPERATOR IN STATE VARIABLE REPRESENTATION.* Variables are written using capitals. The operators shown here correspond to the planning operators from the example 2.4.

A planning operator for ***taking a box on top of a stack by a crane****.*
(p(take), e(take))=
(
{reachable(CRANE)=STACK, holding(CRANE) = nothing,
onTop(STACK) = BOX1, on(BOX2) = BOX1};
{holding(CRANE) = BOX1, onTop(STACK) = BOX2, on(BOX2) = nothing}
)

An action for ***loading the empty truck with a box by a crane at a location****.*
(p(load), e(load))=
(
{holding(CRANE) = BOX, at(TRUCK) = LOCATION, capacity(TRUCK) = two};
{loaded(TRUCK) = BOX, capacity(TRUCK) = one, holding(CRANE) = nothing};
{holding(CRANE) = BOX, capacity(TRUCK) = two}
)

The remaining definitions concerning the planning problem in the state variable representation are analogous to those of classical representation. A planning problem is again a triple (s_0, g, O) where s_0 is the initial state (that is a state in the state variable representation), g is the goal, and O is the set of allowed planning operators. A solution of a planning problem is again a sequence of actions that achieves the goal from the initial state.

The state variable representation is especially useful when the relations among objects in the planning environment have functional character. The expressive power of the classical representation and the representation using state variables is the same.

2.5 Complexity of Planning Problems

We give some complexity results about the planning problems in this section. More results can be found in the literature (Bäckström and Nebel, 1992; Bylander, 1994; Erol *et al.*, 1994b; Erol *et al.*, 1995).

Let us consider a decision version of the planning problem. That is, having a planning problem in the classical representation the task is to determine whether there exists a plan for this problem. Results for the state variable representation are the same.

Proposition 2.1 *(COMPLEXITY OF CLASSICAL PLANNING - PLAN EXISTENCE). Having a planning problem* $P = (s_0, g, O)$ *in the classical representation, the task of determining whether there exists a solution plan for the planning problem* P *is EXPSPACE-complete.* ■

Let us recall that $EXPSPACE = \bigcup_{k \in \mathbb{N}} DSPACE(2^{n^k})$. The complete proof of this theorem is listed in (Ghallab *et al.*, 2004). Sometimes it is convenient to answer the question whether there is a plan of at most a given length. This problem is still difficult.

Proposition 2.2 *(COMPLEXITY OF CLASSICAL PLANNING - BOUNDED PLAN EXISTENCE). Having a planning problem* $P = (s_0, g, O)$ *in the classical representation and* $k \in \mathbb{N}$*, the task is to determine whether there exists a plan for the problem* P *that uses at most* k *actions. This problem is NEXPTIME-complete.* ■

Let us again recall that $NEXPTIME = \bigcup_{k \in \mathbb{N}} NTIME(2^{n^k})$. The deeper discussion of this result is provided in (Ghallab *et al.*, 2004).

All the results presented above are for the worst case. If we have a description of the set of planning operators in advance, it is possible to obtain more optimistic results. To be specific, for the *FREECELL* planning domain from the 2002 International Planning Competition (Long, 2002) the problem of plan existence and the problem of bounded plan existence were proved to be *NP*-complete (Helmert, 2003, 2006).

2.6 Overview of Planning Techniques

There are many algorithmic techniques for solving planning problems. In this section we give an overview of several most successful planning techniques.

First of all, we should mention so called *forward search*. It is backtracking based method that starts from the initial state and exhaustively tries to apply actions to the currently maintained state until a goal is reached (Ghallab *et al.*, 2004).

A simple variation of the forward search is a *backward search* (Ghallab *et al.*, 2004). The backward search works in very similar way as forward search but it starts from the goal and it is trying to reach the initial state. An exhaustive state space exploration is used again.

Although both the forward and backward search guarantee to find a solution if there exists a solution (the algorithm is said to be *complete*) they are extremely inefficient because of the high branching factor at each decision point. There were several attempts to reduce the size of the search space that is necessary to explore by decreasing the branching factor at decision points. The well known *STRIPS* planning algorithm (Fikes and Nilsson, 1971) was such an attempt. The *STRIPS* algorithm was based on backward search but in addition it used several rules how to behave at decision points.

A very successful approach for solving planning problems is based on so called planning graphs. The first algorithm which was based on planning graphs was GraphPlan by Blum and Furst (Blum and Furst, 1997). The search for a plan starts by building the planning graph. The planning graph is a structure which represents the reachability of states. It can be used to answer a question whether it is possible to reach a given state by a given number of actions fast. However, the planning graph is not complete for the reachability questions. If we get the answer that a state is not reachable it is definitely not reachable and it is necessary to allow more actions. If the goal seems to be reachable according to the planning graph then the algorithm tries to extract a plan from the planning graph by search.

The reachability analysis provided by the planning graphs reduces the search space significantly. Many planning systems are based on the idea of state reachability using planning graphs (*IPP* - Koehler *et al.*, 1997; Koehler, 2007; *MaxPlan* - Zhao *et al.*, 2006, 2007; *SATPlan* - Kautz *et al.*, 2006, 2007; *STAN* - Long and Fox, 1999).

Another successful approach for solving planning problems is the usage of Boolean satisfiability. Solvers based on Boolean satisfiability such as *SATPlan* and *MaxPlan* belong among the best solvers according to the results of the last planning competitions *IPC-4* (Edelkamp *et al.*, 2004) and *IPC-5* (Gerevini *et al.*, 2006). The input planning problem is translated automatically into a Boolean formula which is then solved by a solver for Boolean satisfiability such as *MiniSAT* (Eén and Sörensson, 2005, 2007) or *Siege* (Ryan, 2007). The solution of the formula is then translated back to a resulting sequence of actions. The formulation of planning problems as Boolean formulas is often based on planning graphs (Kautz and Selman, 1999).

A very similar approach to the usage of Boolean satisfiability is to translate the planning problem into the formalism of constraint programming. The hand-tailored translation of planning problems into constraint programming formalism is used in *CPlan* planner by van Beek and Chen (1999). This planner is especially successful due to the utilization of reasoning about spatiotemporally distant objects. Another planner that uses constraint programming formalism is *GP-CSP-Plan* of Do and Kambhampati (2000, 2001). This planner is based on translation of planning graphs into the constraint programming formalism.

All the planners mentioned above use some kind of an exhaustive search through the search space. A different approach for solving planning problems is represented by *local search* planners. Local search for a solution is typically based on the exploration of the neighborhood of the currently best found element of the search space (measured according to a certain objective function). If there is a better element in the neighborhood then it is taken as a new best found element. The example of this approach is *Hill-climbing* algorithm (Russell and Norvig, 2003). The representative of the planner based on local search is *LPG-td* (Gerevini and Serina, 2002, 2007). These planners are intrinsically incomplete.

The modern planners extensively use various types of heuristics that are used to guide the search at decision points. Some planners even use heuristics as the main tool for finding a solution. The representative is *HSP* of Bonet and Geffner (2001a, 2001b) or *SGPlan* (Hsu *et al.*, 2006, 2007).

2.7 Planning Using Planning Graphs

The first algorithm based on *planning graphs* - *GraphPlan* (Blum and Furst, 1997) - is able to find concurrent plans of the planning problems (that is, more actions at a certain time step can be performed in parallel, see the plan in figure 2.3). The planning graph is a polynomial size data structure which encapsulates a necessary but insufficient condition for the reachability of a goal. It can be used to answer the question whether it is possible to reach a certain goal by a certain number of actions. If the answer obtained from the planning graph is *yes* then it might be possible but also might not. If the answer obtained from the planning graph is *no* then it is definitely not possible to reach the goal within a given number of steps. This useful property is exploited by the GraphPlan algorithm to effectively cut out large parts of the search space.

The GraphPlan algorithm itself works in two interleaving phases. In the first phase, the planning graph representing the reachability problem for plans up to the certain length is built or extended. In the second phase, the algorithm is trying to extract a valid plan for a given goal from the planning graph. If the second phase is unsuccessful, the algorithm continues with the first phase for plans of extended length.

The extraction of a plan from the planning graph is done by search. The standard chronological backtracking was used for the plan extraction in the original version of the algorithm. Although several other techniques were developed for extracting plans, this phase represents a bottleneck of the whole planning process based on planning graphs (Kambhampati, 2000; Zimmerman and Kambhampati, 2005).

The planning using planning graphs has been developed in many directions since the original contribution by Blum and Furst in 1997. Let us mention the extensions that allow negative preconditions and conditional effects (Kambhampati *et al.*, 1997). Another inter-

esting extension of planning graphs that can handle durative actions is described in (Smith and Weld, 1999).

Although the GraphPlan algorithm is no longer the state-of-the-art algorithm for classical planning, the GraphPlan approach remains the fastest for planning with parallel actions. Moreover, several of today's state-of-the-art heuristically guided planners (Bonet and Geffner, 2001a, 2001b; Nguyen *et al.*, 2002; Gerevini and Serina, 2002) exploit the planning graphs for building powerful heuristics.

This section is devoted to the formal description of the structure of planning graph and GraphPlan algorithm. For simplicity reasons all the formal constructs in this section are defined for the classical representation of planning problems. The reformulation for state variable representation is straightforward. Nevertheless, some examples presented in this chapter (for example figure 2.4) use state variable representation since the state variable representation was used in our experimental evaluation.

2.7.1 Planning Graphs

Assume a fixed language $L = (P_L, C_L, V_L)$ (let us remind that P_L is a finite set of predicate symbols, C_L is a finite set of constants, and V_L is a finite set of variable symbol) associated with the planning environment. In addition, we need so called *no-operation* actions to be able to define planning graphs. For every predicate $p \in P_L$ of arity r_p and constants $c_1, c_2, \ldots, c_{r_p} \in C_L$ we assume a no-operation action $a_{NOOP}^{p(c_1,c_2,\ldots,c_{rp})} = (p(c_1,c_2,\ldots,c_{r_p}), p(c_1,c_2,\ldots,c_{r_p}), \emptyset)$. Let us denote $A_{NOOP} = \{a_{NOOP}^{p(c_1,c_2,\ldots,c_{rp})} \mid p \in P_L;\ c_1, c_2, \ldots, c_{r_p} \in C_L\}$ a set of all no-operations associated with the language L. No-operation is an action that produces exactly what is already present in the state of the planning environment. There is a formal reason for having no-operations in connection with planning graphs - they preserve states between layers of the planning graph.

Another slight restriction is that we do not allow negative literals in the goal in the statement of the planning problem. Observe that this slight restriction is not at the expense of expressivity (it is for instance possible to introduce additional atoms for negated literals). The reason for this restriction is the nature of planning graph construction.

The planning graph for a planning problem $P = (s_0, g, O)$ is defined as follows. It consists of two alternating structures called a *proposition layer* and an *action layer*. The initial state s_0 represents the *0*th proposition layer P_0. The layer P_0 is a set of atoms occurring in the state s_0. The rest of the planning graph is defined inductively. Consider that the planning graph with layers $P_0, A_1, P_1, A_2, P_2, \ldots, A_k, P_k$ has been already constructed (A_i denotes the *i*th action layer, P_i denotes the *i*th proposition layer). The next action layer A_{k+1} consists of all the actions that can be obtained from planning operators in O and of all the no-operations from A_{NOOP} whose preconditions are included in the *k*th proposition layer P_k (previous proposition layer). All the actions added into A_{k+1} must satisfy the additional

condition that no two preconditions of any action are conflicting - in the planning graph terminology conflicting constructs are denoted as *mutually excluded* (if two atoms are mutually excluded, we briefly say that they are *mutex*). The following definitions formalize the notion of mutual exclusion.

Definition 2.13 (INDEPENDENCE OF ACTIONS). A pair of actions $\{a,b\}$ is *independent* if $e^-(a) \cap (p(b) \cup e^+(b)) = \emptyset$ and $e^-(b) \cap (p(a) \cup e^+(a)) = \emptyset$. Otherwise $\{a,b\}$ is a pair of *dependent* actions. □

Definition 2.14 (ACTION MUTEX AND ACTION MUTEX PROPAGATION). We call a pair of actions $\{a,b\}$ within the action layer A_i a *mutex* if either the pair $\{a,b\}$ is dependent or an atom of the precondition of the action a is mutex with an atom of the precondition of the action b (defined in the following definition). □

Definition 2.15 (PROPOSITION MUTEX AND PROPOSITION MUTEX PROPAGATION). We call a pair of atoms $\{p,q\}$ within the proposition layer P_i a *mutex* if every action a within the layer A_i where $p \in e^+(a)$ is mutex with every action b within the action layer A_i for which $q \in e^+(b)$ and the action layer A_i does not contain any action c for which $\{p,q\} \subseteq e^+(c)$. □

The next proposition layer P_{k+1} consists of all the atoms that appear as positive effects of some action in the previous action layer A_{k+1}. We do not care about the negative effects of actions at this point. Finally a set of mutexes is associated with the proposition layer P_{k+1} according to the definition 2.15.

We can now observe the reason for having no-operation actions. They are included in an action layer to preserve already present atoms in the next proposition layer.

Having a planning graph consisting of layers $P_0, A_1, P_1, A_2, P_2, \ldots, A_k, P_k$, we say the planning graph has length k. Having the action layer A_i the set of actions is often identified by the symbol A_i itself and the set of mutexes associated with the *i*th action layer is denoted as μA_i. The similar notation is for proposition layers; having the proposition layer P_i the set of its propositions is identified by the symbol P_i itself and the set of mutexes associated with the *i*th proposition layer is denoted as μP_i. This notation comes from (Ghallab *et al.*, 2004).

Let us now briefly describe how the state reachability analysis is done using the planning graph.

Definition 2.16 (CONCURRENT PLAN). Let the function $\pi^i : \{1,2,\ldots,i\} \to \{1,2,\ldots,i\}$ denotes a permutation of the set $\{1,2,\ldots,i\}$. A concurrent plan for a planning problem $P = (s_0, g, O)$ is a sequence of sets of actions of the form $[\{a_1^1, a_2^1, \ldots, a_{m_1}^1\}; \{a_1^2, a_2^2, \ldots, a_{m_2}^2\}; \ldots; \{a_1^k, a_2^k, \ldots, a_{m_k}^k\}]$ where for every $k-$tuple of permutations

$\pi_1^{m_1},\pi_2^{m_2},\ldots,\pi_k^{m_k}$ the sequence $[a^1_{\pi_1^{m_1}(1)},a^1_{\pi_1^{m_1}(2)},\ldots,a^1_{\pi_1^{m_1}(m_1)};\ a^2_{\pi_2^{m_2}(1)},a^2_{\pi_2^{m_2}(2)},\ldots,a^2_{\pi_2^{m_2}(m_2)};\ldots;$ $a^k_{\pi_k^{m_k}(1)},a^k_{\pi_k^{m_k}(2)},\ldots,a^k_{\pi_k^{m_k}(m_k)}]$ is a solution of the planning problem P. □

The concurrent plan $[\{a_1^1,a_2^1,\ldots,a_{m_1}^1\};\{a_1^2,a_2^2,\ldots,a_{m_2}^2\};\ldots;\{a_1^k,a_2^k,\ldots,a_{m_k}^k\}]$ can be also interpreted in the way that the sets of actions $\{a_1^1,a_2^1,\ldots,a_{m_1}^1\};\{a_1^2,a_2^2,\ldots,a_{m_2}^2\};\ldots;$ $\{a_1^k,a_2^k,\ldots,a_{m_k}^k\}$ respectively can be performed in parallel to achieve the goal. The length of the concurrent plan is defined as the number of sets which the concurrent plan consists of (in our case it is k).

Definition 2.17 (CONCURRENT PLAN DETERMINED BY PLANNING GRAPH). A concurrent plan $[\{a_1^1,a_2^1,\ldots,a_{m_1}^1\};\{a_1^2,a_2^2,\ldots,a_{m_2}^2\};\ldots;\{a_1^k,a_2^k,\ldots,a_{m_k}^k\}]$ for a planning problem $P=(s_0,g,O)$ is said to be *determined by the planning graph* consisting of layers $P_0,A_1,P_1,A_2,P_2,\ldots,A_k,P_k$ if $\{a_1^i,a_2^i,\ldots,a_{m_i}^i\}\subseteq A_i$ is a pair-wise non-mutex set of actions with respect to μA_i for every $i=1,2,\ldots,k$. □

Proposition 2.3 *(NECESSARY CONDITION ON STATE/GOAL REACHABILITY). Let us have a planning graph consisting of layers $P_0,A_1,P_1,A_2,P_2,\ldots,A_k,P_k$ for a planning problem $P=(s_0,g,O)$. If $g\not\subseteq P_k$ or if $g\subseteq P_k$ and g is not mutex-free with respect to μP_k then the problem P cannot be solved by any concurrent plan of length k.* ■

We omit details of the proof of this proposition since it is given in (Blum and Furst, 1997) with appropriate analysis. It gives the necessary condition on the existence of a solution of a given planning problem. Notice that the proposition can be applied on reachability of states as well as goals since they are both sets of atoms. If we construct a planning graph with a certain number of layers, then a given goal (or a state) must be contained in the last proposition layer and must be mutex-free (there is no mutex between any pair of atoms of the goal in the last proposition layer). More precisely, if this condition does not hold (an atom of the goal is not contained in the last proposition layer or some two atoms of the goal are mutex with respect to the last proposition layer) then the given goal cannot be reached by any concurrent plan of the length that equals to the length of the planning graph.

Before continuing with the algorithms concerning planning graphs let us briefly discuss the computational resources required by planning graphs. The following results can be found in (Ghallab *et al.*, 2004).

Proposition 2.4 (SPACE COMPLEXITY OF PLANNING GRAPHS). *The space required by an action layer of the planning graph is polynomial in size of the input planning problem. The space required by a proposition layer of the planning graph is also polynomial in size of the input planning problem.* ■

Proof. The size of the input problem is the length of the complete description necessary for the language $L = (P_L, C_L, V_L)$ and for the problem instance $P = (s_0, g, O)$.

First let us estimate the number of all possible ground atoms that can appear within the proposition layer of the planning graph. Let α be the upper bound on the number of parameters of any operator in O. Let us denote $r = \max_{p \in P_L} arity(p)$ ($arity(p)$ is the arity of the predicate symbol p). The number of all possible ground atoms is bounded by $|P_L||C_L|^r$.

The total number of actions that can form the action layer A_i including no-operation actions is bounded by $|C_L|^\alpha |O| + |A_{NOOP}|$, where $|A_{NOOP}| \le |P_L||C_L|^r$. The number of action mutexes associated with the action layer A_i is bounded by $|A_i|^2 = (|C_L|^\alpha |O| + |P_L||C_L|^r)^2$ since there can be a mutex at most between every pair of actions.

The total number of propositions within the proposition layer P_i is bounded by $|P_L||C_L|^r$. Hence the number of all mutexes associated with the proposition layer P_i is bounded by $|P_i|^2 \le |P_L|^2 |C_L|^{2r}$. ■

The important property of a planning graph for a given problem is that is has a *fixed point level* which is the smallest κ such that for every i, $\kappa < i \le k$, ith proposition and action layers are identical to respective κth proposition and action layers (that is $P_\kappa = P_i$, $\mu P_\kappa = \mu P_i$, $A_\kappa = A_i$, and $\mu A_\kappa = \mu A_i$). Therefore it is useless to build planning graph of the length beyond the fixed point level. The complexity of planning graphs is summarized in the table 2.1.

Planning graph	Space requirements
***i*th action layer**	$\|A_i\| \le \|C_L\|^\alpha \|O\| + \|P_L\|\|C_L\|^r$ $\|\mu A_i\| \le (\|C_L\|^\alpha \|O\| + \|P_L\|\|C_L\|^r)^2$, where $r = \max_{p \in P_L} arity(p)$, α is the upper bound on number of parameters of operators
***i*th proposition layer**	$\|P_i\| \le \|P_L\|\|C_L\|^r$ $\|\mu P_i\| \le \|P_L\|^2 \|C_L\|^{2r}$, where $r = \max_{p \in P_L} arity(p)$

Table 2.1. *COMPLEXITY OF PLANNING GRAPH.* Summary of space requirement of the structure of planning graph.

When implementing the data structures for the planning graph it is convenient to store relationships between action and proposition layers. In particular, it is convenient to store the support relations between an action and atoms from the next proposition layer that are positive effects of this action (we also say that the action is support for these atoms). This support relation is represented as a set of links between the action and atoms from the next proposition layer. The next relation that is often stored within the planning graph is a precondition relation between an action and atoms from the previous proposition layer. More precisely, this relation captures information of what atoms from the proposition layer are

the preconditions for the action. This precondition relation is represented by links between action's precondition atoms from the proposition layer and the corresponding action.

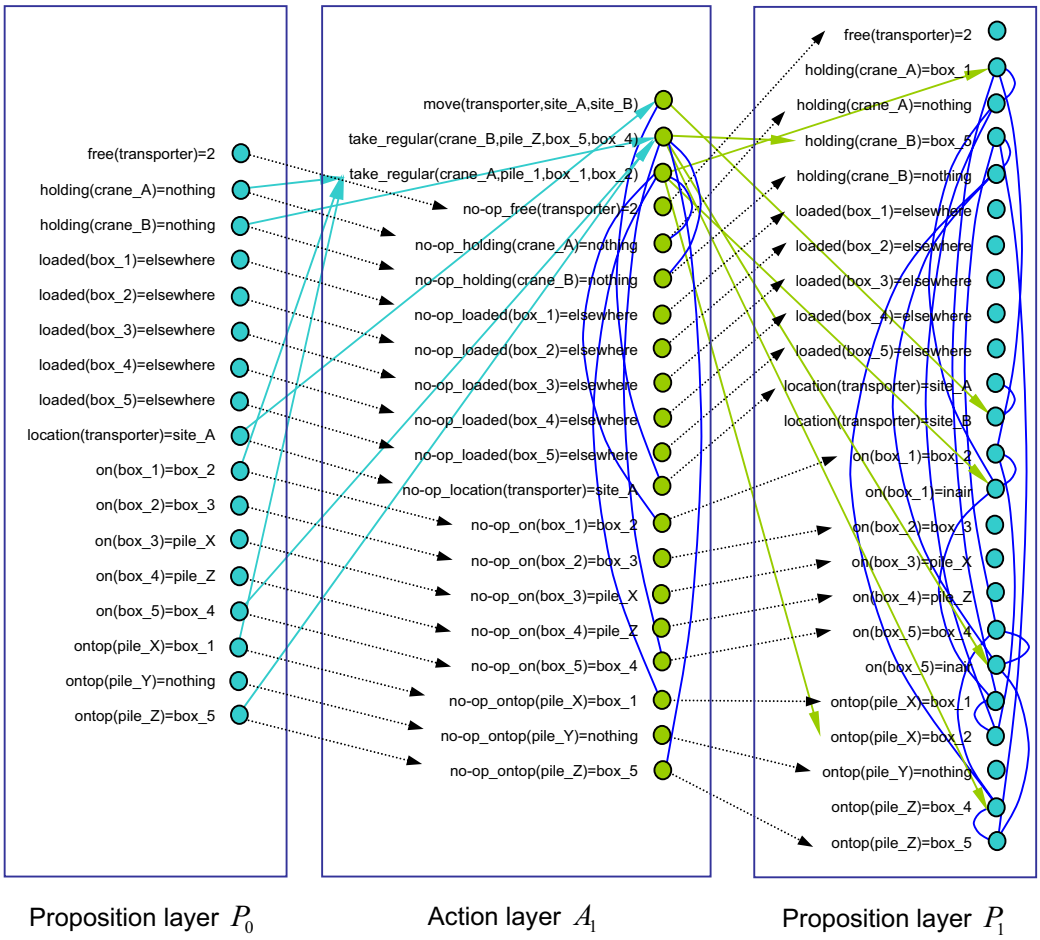


Figure 2.4. *STRUCTURE OF PLANNING GRAPH.* First three layers of the planning graph for the dock worker robots planning problem depicted in figures 2.1, 2.2, and 2.3. The depicted planning graph uses state variable representation. Support and precondition relations are represented by green and blue arrows respectively (relations between no-operation actions are depicted using dotted arrows). Proposition and action mutexes are represented by blue arcs.

An example of the planning graph using the state variable representation is shown in figure 2.4. The advantage of the state variable representation is that assignments of the same state variable within the proposition layer are automatically pair-wise mutex. This property allows slightly more effective implementation (there is no need to explicitly store mutexes between the assignments to a single state variable). We use the state variable representation in the experimental implementation for our empirical evaluation.

2.7.2 GraphPlan Algorithm

The GraphPlan algorithm is a planning algorithm based on state reachability analysis using the data structure of planning graphs. The algorithm consists of two interleaved phases. The first phase is represented by *incremental expansion* of the planning graph. The incremental expansion of the planning graph is just adding one action and one proposition layer respectively as new last two layers of the planning graph (notice that it is also possible to expand the planning graph with more than one action and proposition layers).

The second phase of the GraphPlan algorithm consists in an attempt to *extract a concurrent plan* from the planning graph. This phase is performed only if the given goal is reachable in the constructed planning graph. If the plan extraction is unsuccessful then the algorithm continues with the planning graph expansion phase. These two phases are repeated till the planning problem is solved or a termination condition is reached.

The key parts of the basic version of the GraphPlan algorithm are shown here as algorithms 2.1 and 2.2. The algorithm 2.1 formally describes the plan extraction phase. Algorithm 2.2 represents the main loop which repeatedly performs the planning graph expansion and the extraction phases and checks the termination condition.

The presented symbolic code will be used for all the algorithms throughout the thesis. We use procedures (does not have a return value) and functions (has a return value) as the main structural entities. The control structure of procedures and functions is depicted using indentation. The symbolic code omits the internal details of data structures as it is common in the literature.

Let us now concentrate on the plan extraction phase. Suppose we have a goal for which we are trying to find a concurrent plan starting in the initial state. Next suppose that we have constructed the planning graph for the given planning problem up to a certain length. The process of search for a concurrent plan starts with satisfying the goal at the last layer of the planning graph. First, it is checked if the given goal is contained in the last proposition layer (all the goal's atoms must occur in the last proposition layer) and if it is mutex-free with respect to the layer (there must be no mutex between any atoms of the goal with respect to the last proposition layer). If this condition does not hold, the process terminates with the answer that the goal cannot be satisfied. Otherwise the process continues by finding a set of mutex-free actions from the last action layer (the layer preceding the last proposition layer) that satisfy the goal (we also say these actions support the goal). This set of supporting actions induces another goal. This goal is formed by preconditions of the actions from the set of supporting actions. The plan extracting procedure is recursively called at this point with parameters specifying the new goal and the intention to extract this goal from the previous layer. If the recursive call of the procedure is unsuccessful the algorithm continues with further attempts to find another set of non-mutex actions supporting the original goal. If the recursive call is successful, we have a concurrent plan.

Algorithm 2.1. *GRAPHPLAN - PLAN EXTRACTION*. Plan extraction phase of the GraphPlan algorithm. The data structure of the planning graph is a compound structure consisting of several arrays indexed by layers - array of sets of propositions, array of sets of proposition mutexes, array of sets of actions, array of sets of action mutexes, and array of sets of nogoods. The individual components of the planning graph data structure are accessed using . ("dot") in the symbolic code.

function $extractPlan(pG,l,g)$: **sequence**
1: **if** $l = 0$ **then**
2: | **if** $g \subseteq pG.P[0]$ **then return** []
3: | **else return** [*failure*]
4: **if** g is subsumed in $pG.N[l]$ **then return** [*failure*]
5: $\xi \leftarrow extractPlanFromLayer(pG,l,g,\emptyset)$
6: **if** $\xi = [failure]$ **then**
7: | $pG.N[l] \leftarrow pG.N[l] \cup \{g\}$
8: | **return** [*failure*]
9: **else return** ξ

function $extractPlanFromLayer(pG,l,g,\zeta)$: **sequence**
10: **if** $g = \emptyset$ **then**
11: | $g' \leftarrow \{p(a) \mid a \in \xi\}$
12: | $\Xi \leftarrow extractPlan(pG,l-1,g')$
13: | **if** $\Xi = [failure]$ **then return** [*failure*]
14: | **else return** $concatenate(\Xi,\xi)$
15: **else**
16: | select any $t \in g$
17: | $s_t \leftarrow \{a \mid a \in pG.A[l] \,\&\, t \in e^+(a)\}$
18: | **if** $s_t = \emptyset$ **then return** [*failure*]
19 | **for each** $a \in s_t$ **do**
20: | | **if** $checkSupports(pG,l,\xi,a)$ **then**
21: | | | $g' \leftarrow g - e^+(a)$
22: | | | $\xi' \leftarrow \xi \cup \{a\}$
23: | | | **return** $extractPlanFromLayer(pG,l,g',\xi')$
24: **return** [*failure*]

function $checkSupports(pG,l,a,\zeta)$: **boolean**
25: **for each** $b \in \xi$ **do**
26: | **if** $\{a,b\} \in pG.\mu A[l]$ **then return** *False*
27: **return** *True*

Observe that the decision point at which the above search procedure branches is determining which actions from the action layer will support a goal. In the next sections we will describe how this process can be improved.

The algorithm 2.1 is a formalization of the above search process. It consists of three functions *extractPlan*, *extractPlanFromLayer*, and *checkSupports*. The function *extractPlan* represents the top calling function. This function constructs a concurrent plan for a given goal and for a given planning graph of a given length (number of layers). If search for a concurrent plan is successful, the function returns the concurrent plan as a sequence of sets. Otherwise a singleton sequence containing the symbol *failure* is returned. The next two functions are auxiliary. The function *extractPlanFromLayer* finishes the concurrent plan for a given goal at a given layer of the planning graph supposed that a certain set of actions has been already selected into the plan. Finally, the function *checkSupports* checks whether an action can be added to the set of actions such that the resulting set of actions is mutex-free.

The function *extractPlan* gets three parameters - pG, l, and g. The parameter pG is the structure of planning graph. The parameter l is the length of the planning graph and the parameter g is the goal we want to satisfy. The structure of planning graph pG is a compound structure which consists of several arrays of sets representing layers of the planning graph. The components of this compound structure are accessed using the "." (dot) notation. The components are following: $pG.P$ is an array indexed by layers whose cells are sets representing atoms of the individual proposition layers, $pG.\mu P$ is an array indexed by layers representing proposition mutexes of the individual proposition layers; $pG.A$ and $pG.\mu A$ are arrays again indexed by layers representing actions and action mutexes respectively of individual action layers, and finally $pG.N$ is an array indexed by layers containing so called proposition *nogoods* for the individual proposition layers.

A nogood associated with a certain proposition layer of the planning graph is a set of atoms that cannot be satisfied together in the proposition layer (even though they are mutex-free). The nogoods are inferred in course of search for a concurrent plan. If it is found in course of search that a certain goal cannot be satisfied in a certain proposition layer, the goal is declared to be a nogood. If later another attempt to satisfy a goal containing this nogood as a sub-goal (nogood is a subset of the goal) occurs at the same proposition layer, it is possible to identify the goal as unsatisfiable without search by checking in against nogoods associated with the layer.

The call $extractPlan(pG,l,g)$ attempts to extract a concurrent plan from the planning graph pG to satisfy the goal g in the lth proposition layer. If we are satisfying the goal in the initial layer it is sufficient to check whether g is contained in the initial layer (lines 1-3). If this is not the case and we are somewhere within the planning graph, a check of the goal g against stored nogoods stored for the lth proposition layer is performed (line 4). If the goal g is not subsumed by the stored nogoods, the function *extractPlanFromLayer* is called with parameters specifying that we need to extract a concurrent plan for the goal g

from the planning graph pG starting in the *l*th proposition layer and no action has been selected yet ($\emptyset$ as the last parameter - line 5). If the concurrent plan extraction is successful we are finished, otherwise a new nogood for the specified layer is stored (lines 6-9). The call $extractPlanFromLayer(pG,l,g,\zeta)$ specifies that a concurrent plan should be extracted from the planning graph pG starting at the *l*th proposition layer and supposed that we have already included actions listed in the set ζ into the *l*th set of actions of the concurrent plan and the partial goal g remains to be satisfied. The function starts by testing if the remaining goal g is empty (line 10). If this is the case a new goal g' for the previous proposition layer is formed by the union of all the preconditions of all the actions from the set ζ (line 11). Then the function *extractPlan* is called with parameters specifying that a concurrent plan for the goal g' should be extracted from the planning graph starting in the *(l-1)*th proposition layer (line 12). Finally, in this execution branch a test whether the plan extraction was successful or not is performed (line 13-14). If it is successful ζ is added as the last member of the constructed concurrent plan sequence otherwise a failure is reported to the superior calling procedure. If there is non-empty partial goal g (lines 15-24) an atom t is selected from the goal (line 16) and a set of actions s_t from the *l*th action layer which has atom t as its positive effect is constructed (s_t is called a set of supports for atom t - line 17). If there is no supporting action for atom t then the function reports a failure (line 18) otherwise the search for a supporting action which forms a mutex-free set of actions together with actions from ζ starts (lines 19-23). Each candidate action a from s_t is checked if it forms together with ζ a mutex-free set. This check is implemented by the function *checkSupports* which gets parameters pG, l, a, and ζ. If $\zeta \cup \{a\}$ forms a mutex-free set with respect to the *l*th action layer of the planning graph pG the *True* value is returned (lines 25-27). If the check is successfully passed a new partial goal g' and a new set of actions ξ' are passed to the subsequent call of the *extractPlanFromLayer* function (lines 21-23). The new goal g' and the new set of actions ξ' reflect the fulfilled decision - a new action is selected as a member of the concurrent plan sequence which shrinks the goal yet to be satisfied. This process can be also viewed as transfer of the goal g into the set of mutex-free supporting actions ζ and continuing to the lower layer at the moment the whole goal is supported. Search for the set of mutex-free set of supporting actions for a goal represents one of the main bottlenecks of the GraphPlan algorithm.

The algorithm 2.2 represents the top control loop of the algorithm which repeatedly executes planning graph expansion and concurrent plan extraction phases. The algorithm consists of two functions - *generatePlanGP* and *checkFixedPoint*. The function *generatePlanGP* is the main control function; it gets parameters s_0, g, and O specifying the problem. The return value of the function is a sequence representing concurrent plan for the planning problem (s_0,g,O). The function *checkFixedPoint* with parameters pG and l checks whether a specified planning graph pG of a given length l reached its fixed point.

Algorithm 2.2. *GRAPHPLAN.* Main loop of the GrapPlan planning algorithm. The code listed here uses an external function *expandPlanningGraph* which expands a specified planning graph up to a specified length; the resulting planning graph is returned as a return value.

function $generatePlanGP(s_0, g, O)$: **sequence**
1: $pG.P[0] \leftarrow s_0$
2: $pG.\mu P[0] \leftarrow \emptyset$
3: $pG.N[0] \leftarrow \emptyset$
4: $\kappa \leftarrow \perp$
5: $\mathrm{K} \leftarrow 0$
6: $l \leftarrow 0$
7: **until** $(g \subseteq pG.P[l]$ **and** $g^2 \cap pG.\mu P[l] = \emptyset)$ **or** $checkFixedPoint(pG, l)$ **do**
8: $\quad l \leftarrow l + 1$
9: $\quad pG \leftarrow expandPlanningGraph(pG, l)$
10: **if** $g \not\subseteq pG.P[l]$ **or** $g^2 \cap pG.\mu P[l] \neq \emptyset$ **then return** $[failure]$
11: **if** $checkFixedPoint(pG, l)$ **then**
12: $\quad \kappa \leftarrow l$
13: $\quad \mathrm{K} \leftarrow |pG.N[l]|$
14: **do**
15: $\quad \Xi \leftarrow extractPlan(pG, l, g)$
16: $\quad$ **if** $\Xi = [failure]$ **then**
17: $\quad\quad$ **if** $checkFixedPoint(pG, l)$ **then**
18: $\quad\quad\quad$ **if** $\kappa = \perp$ **then** $\kappa \leftarrow l$
19: $\quad\quad\quad$ **if** $\mathrm{K} = |pG.N[\kappa]|$ **return** $[failure]$
20: $\quad\quad\quad \mathrm{K} \leftarrow |pG.N[\kappa]|$
21: $\quad\quad l \leftarrow l + 1$
22: $\quad\quad pG \leftarrow expandPlanningGraph(pG, l)$
23: **while** $\Xi = [failure]$
24: **return** Ξ

function $checkFixedPoint(pG, l)$: **boolean**
25: **if** $l \leq 0$ **then return** *False*
26: **else**
27: $\quad$ **if** $pG.P[l-1] \neq pG.P[l]$ **or** $pG.\mu P[l-1] \neq pG.\mu P[l]$ **then return** *False*
28: $\quad$ **else**
29: $\quad\quad$ **if** $pG.A[l-1] \neq pG.A[l]$ **or** $pG.\mu A[l-1] \neq pG.\mu A[l]$ **then return** *False*
30: **return** *True*

The function *generatePlanGP* builds the planning graph for the given planning problem up to the level where the given goal is contained or the fixed point is reached (lines 1-9). Within this process the function *expandPlanningGraph* is used. This function expands a specified planning graph pG up to a specified length l. In our case, the function just adds one action and one proposition layer to the existing planning graph. Notice the setup of the fixed point level κ and its size K (lines 4-5, 12-13). This is important for termination condition. Notice also, that if the fixed point is reached and the goal is still not contained in the last proposition layer or it is not mutex-free, the algorithm terminates with failure (line 10).

After the initial planning graph construction, the algorithm continues by a loop in which plan extraction and planning graph expansion interleave (lines 14-23). An attempt to extract a concurrent plan starting in the currently last proposition layer of the planning graph satisfying the given goal is performed first (line 15). If the plan extraction is unsuccessful, the termination condition is checked (lines 16-20) and planning graph is expanded by one action and proposition layer (lines 21-22). The algorithm terminates with failure if the set of nogoods in the fixed point level κ does not change after two subsequent unsuccessful plan extractions (more precisely, all the attempts to satisfy a goal in the fixed point level are caught within the check against the set of recorded nogoods). If the plan extraction is successful, the function returns a solution concurrent plan (line 24).

2.8 Constraint Programming

Constraint programming provides a framework for modeling and solving a large variety of problems arising in combinatorial optimization, artificial intelligence, computer graphics, planning, and scheduling etc. The pivotal concept in constraint programming is to separate problem modeling and problem solving. The problem is modeled using concepts such as a variable and a constraint which is a relation between the variables that we want to satisfy. The solving techniques range from intelligent exhaustive search (Baker, 1995) to various local techniques based on greedy exploration of the search space (Harvey, 1995).

One of the most successful techniques developed within constraint programming are so called *consistency techniques*. A consistency technique allows reducing the search space significantly by a clever pruning of the sets of possible values for the variables.

However, even with the usage of consistency techniques and intelligent search there is still a lot of problems that are too difficult to be solved by these techniques only. Therefore problem oriented heuristics are used to guide the search for a solution. Authors of software products offering constraint technology such as *ILOG* (ILOG, 2007), *CHOCO* (CHOCO, 2007), *Gecode* (Gecode, 2007), and *SICStus Prolog* (SICStus, 2007) are aware of this fact and provide large collections of built-in problem oriented heuristics.

2.8.1 Constraint Satisfaction Problems

The *constraint satisfaction problem (CSP)* consists of *variables* and of *constraints*. Each variable has assigned a finite *domain* of values for it. A constraint is an arbitrary relation over the domains of the variables in the scope of the constraint. Constraints represent relations between variables and restrict the values of the variables to allowed combinations only. The number of variables involved in the constraint is called an *arity* of the constraint.

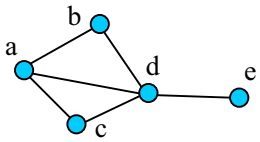
Definition 2.18 (CONSTRAINT SATISFACTION PROBLEM). A *constraint satisfaction problem* is a triple (X, C, D) where X is a finite *set of variables* and C is a finite *set of constraints* over the variables from the set X, and each variable from X is assigned *domain* of values D. A sequence of variables constrained by a constraint $c \in C$ is called a *scope* of the constraint and will be denoted as X_c (the order of variables in the scope matters). □

It is convenient to assign a so called *current domain* of values to each variable. The current domain for variable $x \in X$ is denoted as $D[x]$. Current domains are initially equal to D and they are used as working domains by algorithms.

Example 2.7 shows a formulation of the graph coloring problem as a constraint satisfaction problem. Let us have a graph $G = (V, E)$ and a number $k \in \mathbb{N}$ which determines the maximum number of colors that can be used. The task is to assign a color to each vertex of the graph such that no two adjacent vertices have the same color.

Example 2.7. *EXAMPLE OF A CONSTRAINT SATISFACTION PROBLEM.* The graph coloring problem expressed as a constraint satisfaction problem.

G = (V, E); k = 3



Corresponding constraint satisfaction problem P = (X, C)
X = {a, b, c, d, e}
a, b, c, d, e ∈ {1, 2, 3}
C = {a ≠ b, a ≠ c, b ≠ d, c ≠ d, d ≠ e}

To solve a given constraint satisfaction problem it is necessary to assign values to variables such that all the constraints are satisfied. The following definitions formally describe the solution of a constraint satisfaction problem.

Definition 2.19 (ASSIGNMENT OF VALUES TO VARIABLES). A *complete assignment* of values to variables for the problem $P = (X, C)$ is a function $\psi : X \to D$. A partial assignment of values to variables for the problem $P = (X, C)$ is a function $\varphi : X' \to D$ where $\emptyset \neq X' \subset X$. □

Definition 2.20 (SOLUTION OF CONSTRAINT SATISFACTION PROBLEM). A *solution* of a given constraint satisfaction problem $P = (X, C)$ is a complete assignment of values to variables $\psi : X \to D$ and $(\forall c \in C)[x_1, x_2, \ldots, x_k] = X_c \Rightarrow [\psi(x_1), \psi(x_2), \ldots, \psi(x_k)] \in c$ (constraint is a subset of the Cartesian product of the domains of variables of its scope). □

One of the solutions of the problem from example 2.7 is the following complete assignment $\psi(a) = 1$, $\psi(b) = 2$, $\psi(c) = 2$, $\psi(d) = 3$, and $\psi(e) = 1$.

The complexity of solving constraint satisfaction problems is summarized in the following proposition.

Proposition 2.5 (COMPLEXITY OF CONSTRAINT SATISFACTION PROBLEMS). *Having a constraint satisfaction problem P the problem of deciding whether there is a solution of the problem P is NP-complete.* ■

This proposition can be easily proved by using polynomial time transformation of some *NP*-complete problem to a constraint satisfaction problem. As it was shown in the example 2.7 the graph coloring problem which is a known *NP*-complete problem can be used for this.

2.8.2 Solving Techniques

There is a wide variety of techniques for solving constraint satisfaction problems (Dechter, 2003). Techniques for solving CSPs range from *backtracking-based search* to *local search* (Baker, 1995; Harvey, 1995). In the thesis we are focusing on backtracking-based methods only. The reason is that these methods are able to prove non-existence of a solution (the search space is explored systematically; nothing is skipped) and this is a property we need in our applications of CP.

The core of all the solving algorithms discussed in the thesis is formed by *chronological backtracking*. The algorithm performs a systematic search through the set of all possible assignments of values to the variables. At each step the algorithm assigns a value to the variable and checks satisfaction of the constraints by this assignment. If the assignment satisfies the constraints, the next variable is assigned. Otherwise another value is tried for the variable. More precisely, the task is to assign values to all the unassigned variables such that all the constraints are satisfied. The task is solved recursively. At each level of the recursion, the algorithm heuristically selects a variable that will be assignment a value (a *variable ordering heuristic* is used for this selection). Then all the values from the variable's current domain are systematically tried (the order of values is determined by a *value ordering heuristic*). That is, a selected value is temporarily assigned to the variable and satisfaction of constraints is checked (it is sufficient to check satisfaction of the constraints

affected by the assignment; that is, only constraints containing the newly assigned variable in their scope need to be checked). If the constraints are satisfied, we obtained a task of the same type but smaller. Thus the algorithm recursively proceeds with the next unassigned variable. If the constraints are not satisfied or if the smaller task cannot be solved, the variable is unassigned and the next value from its current domain is tried. If all the values for the variable were tried and the solution has not yet been found, the algorithm returns to the previous variable and tries the next value for it. The algorithm terminates with success when all the variables are successfully assigned.

The basic chronological backtracking algorithm is inefficient. Its worst case time complexity is exponential in the size of the CSP ($O(|D|^{|X|})$). Although it is unlikely (unless $P = NP$) to have a solving algorithm with lower worst case time complexity than exponential, there is a large room for improvements.

Again there is a variety of techniques for improving backtracking-based search algorithms. In the following text, we discuss only selected approaches that are most important with respect to our own contributions - consistency techniques, global constraints, and problem modeling.

2.8.3 Consistency Techniques

A *consistency technique* is used for ruling out *inconsistent* values or the tuples of values from further consideration. If the value or the tuple of values is inconsistent then it cannot be part of any solution of the problem.

Consider the following simple example. Let us have two variables $a \in \{1,2\}$, $b \in \{1\}$ and a constraint $a \neq b$ in the constraint satisfaction problem. Then we can immediately rule out the value 1 from the current domain of variable a since it cannot be part of any solution of the problem.

Depending on the strength of a consistency technique it is possible to rule out more or less values (or tuples of values) from the current domains of variables. However, the stronger the consistency is the more time and space is required to establish it in the problem. Therefore a trade-off is necessary to be found between the strength of the consistency and its requirements of computational resources.

The main area of application of consistency techniques are general search algorithms for solving constraint satisfaction problems. The early removal of inconsistencies from the problem saves the time which the search algorithm would consume by unsuccessful attempts to assign inconsistent values to the variables otherwise. In many search algorithms a certain type of consistency is enforced in the problem after each decision of the algorithm (Jussien *et al.*, 2000; Surynek, 2003; Surynek, 2005).

2.8.4 Arc-consistency

Arc-consistency combines simplicity of implementation with the relatively strong pruning power (Mackworth, 1977). Arc-consistency is a technique that removes values (not tuples of values) from the current domains of variables. The original definition of arc-consistency was formulated for problems consisting of binary constraints only. For simplicity we use the restriction on binary constraints too. In the following definition a term *arc* refers to the ordered pair of variables.

Definition 2.21 (ARC-CONSISTENCY FOR BINARY CONSTRAINTS). Consider a binary constraint satisfaction problem $P = (X, C, D)$ and a binary constraint $c \in C$ where $X_c = [x, y]$. An arc $[x, y]$ is *arc-consistent* with respect to the constraint c if for each value $d_x \in D[x]$ there is a value $d_y \in D[y]$ such that an assignment $x = d_x$, $y = d_y$ satisfies the constraint c (that is $[d_x, d_y] \in c$). The binary constraint $c \in C$, where $X_c = [x, y]$, is *arc-consistent* if the arcs $[x, y]$ and $[y, x]$ are arc-consistent with respect to the constraint c. We say the problem $P = (X, C, D)$ to be *arc-consistent* if all the constraints in the set C are arc-consistent. □

Observe that the values that are inconsistent with respect to arc-consistency cannot participate in any solution. Therefore removal of inconsistent values with respect to arc-consistency does not change the set of solutions of the problem.

The task of enforcing arc-consistency is to compute the maximum (with respect to inclusion) current domains of variables such that the problem is arc-consistent.

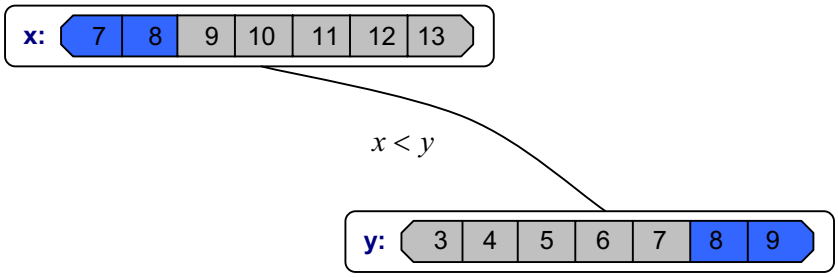


Figure 2.5. *ARC-CONSISTENCY OF A BINARY CONSTRAINT.* Original and current domains of variables x and y are depicted in the figure. Values removed from the current domains are depicted by the grey color. The current domains of variables are represented by the blue color.

Making a single constraint arc-consistent is easy. It is possible to exactly follow the definition of arc-consistency and rule out values form the domains of the corresponding variables. The arc-consistent state of the constraint $x < y$ is shown in figure 2.5. Enforcing arc-consistency for the whole constraint satisfaction problem is more difficult. Making a

certain constraint arc-consistent by removing values from the domains of its variables may cause that another constraint may become inconsistent.

The most straightforward way to make the problem arc-consistent is to make all the constraints arc-consistent and repeat this process until current domains of variables are changing. This is the way how the basic algorithm *AC-1* proceeds. An improvement of AC-1 is represented by the algorithm *AC-2* (Waltz, 1975). A further improvement is represented by the algorithm *AC-3* (Mackworth, 1977).

2.8.5 Algorithm AC-3

AC-3 algorithm is used as a consistency enforcing procedure in planning graphs therefore, a more detailed description is devoted to this algorithm. The idea of the algorithm is to repeatedly make constraints arc-consistent until the current domains of variables are changing. The symbolic code of the algorithm is shown here as algorithm 2.3.

Algorithm 2.3. *AC-3.* Basic algorithm for enforcing arc-consistency.

procedure *propagateAC-3* (C_{REVISE})
1: $Q_{REVISE} \leftarrow C_{REVISE}$
2: **while** $Q_{REVISE} \neq \emptyset$ **do**
3: | $c \in Q_{REVISE}$, arbitrary constraint
4: | $Q_{REVISE} \leftarrow Q_{REVISE} - \{c\}$
5: | $[x, y] \leftarrow X_c$
6: | $[D_x^-, D_y^-] \leftarrow filter(c, D[x], D[y])$
7: | **if** $D_x^- \neq \emptyset$ **then**
8: | | $D[x] \leftarrow D[x] - D_x^-$
9: | | $Q_{REVISE} \leftarrow Q_{REVISE} \cup \{e \in C \mid x \in X_e \,\&\, e \neq c\}$
10: | **if** $D_y^- \neq \emptyset$ **then**
11: | | $D[y] \leftarrow D[y] - D_y^-$
12: | | $Q_{REVISE} \leftarrow Q_{REVISE} \cup \{e \in C \mid y \in X_e \,\&\, e \neq c\}$

The symbolic code of the algorithm AC-3 builds on the function *filter* which makes a single constraint arc-consistent. The function gets the constraint and current domains of the variables bound by the constraint as its parameters. It works exactly according to the definition of arc-consistency. The return value of the function *filter* is an ordered pair of sets of values which have to be removed from the current domains of variables bound by the constraint to make the constraint arc-consistent.

The algorithm consists of a single procedure *propagateAC-3* which has a set of constraints that we want to make arc-consistent as a parameter. To make the whole constraint

satisfaction problem $P=(X,C,D)$ arc-consistent the procedure should be invoked by the call *propagateAC-3*(C). The procedure itself is represented by a loop (lines 2-12) controlled by the queue of constraints which should be revised for arc-consistency. In each iteration of the loop a constraint is selected from the queue (lines 3-4) and values which have to be ruled out from the current domain to enforce arc-consistency are found (lines 5-6). Then the current domains of variables bound by the revised constraint are updated (values are removed - lines 7-9 and 10-12). When doing this, newly affected constraints by the change of the current domains are scheduled into the queue of constraints for revision (lines 9 and 12).

The complexity of the AC-3 algorithm depends on the effectiveness of the implementation of the *filter* function. Assuming the very basic implementation (sequential search for the consistent value in the current domain of the neighboring variable through the constraint), the worst case time complexity of the *filter* function is $O(|D[x]||D[y]|)$ when called for the constraint over the variables x and y. The worst case time complexity of the whole algorithm is summarized in the following proposition. We omit the detailed proof of the proposition since it is given in (Mackworth, 1977) and summarized in (Surynek, 2003).

Proposition 2.6 (WORST CASE TIME COMPLEXITY OF AC-3). *Assuming the basic implementation of the function filter (with the worst case time complexity of* $O(|D[x]||D[y]|)$ *over the constraint with variables* x *and* y*) the AC-3 algorithm for enforcing arc-consistency in the problem* $P=(X,C,D)$ *has the worst case time complexity of* $O(\sum_{c\in C, X_c=[x,y]}(|D[x]|+|D[y]|)|D[x]||D[y]|)$ *which is* $O(|C||D|^3)$. ■

Idea of proof. The key idea of the proposition is the observation that function *filter* can be invoked at most $\sum_{c\in C, X_c=[x,y]}(|D[x]|+|D[y]|)$ times throughout the loop considering that each call the function removes only one value from a current domain of a variable. ■

The space required by the algorithm is proportional to the size of the queue of constraints for revision which is proportional to $|C|$. The complexity of the algorithm is summarized in table 2.2.

There exist more efficient algorithms for enforcing arc-consistency than AC-3. The several best known are *AC-3.1* (Zhang and Yap, 2001), *AC-4* (Mohr and Henderson, 1986), *AC-6* (Bessière and Cordier, 1993), and *AC-2000/AC-2001* (Bessière and Régin, 2001). The common characteristic of these algorithms is a more effective implementation of the *filter* function. Time complexities of these algorithms are lower than that of AC-3, nevertheless the space complexity is typically higher since these algorithms use advanced data structures such as support counters to speedup the search for a consistent value. The more detailed summary of the collection of these arc-consistency enforcing algorithms is given in (Surynek, 2003; Surynek, 2005).

AC-3	Propagation
Space complexity (worst case)	$O(\|C\|)$
Time complexity (worst case)	$O(\|C\|\|D\|^3)$ $O(\sum_{c\in C, X_c=[x,y]}(\|D[x]\|+\|D[y]\|)\|D[x]\|\|D[y]\|)$

Table 2.2. *COMPLEXITY OF AC-3.* Summary of asymptotic space and time complexity of the AC-3 algorithm.

2.8.6 Global Constraints

Consider a situation where we have a set of n variables $x_1, x_2, \ldots, x_n$ with certain domains and we require that all these variables have assigned pair-wise different values. This relation can be modeled by $n(n+1)/2$ simple constraints $x_i \neq x_j$ for $i, j = 1,2,\ldots,n \wedge i \neq j$. However, an explicit representation of the relation was lost because the original relation was fragmented into many simple constraints. By losing the explicit formulation of the relation we also lost the ability to exploit this explicit formulation for stronger consistency enforcing over the variables participating in the relation (for instance arc-consistency does not identify any value as inconsistent in the following CSP though it has no solution: $P = (X, C, D)$, where $X = \{x_1, x_2, x_3\}$, $C = \{x_1 \neq x_2, x_2 \neq x_3, x_1 \neq x_3\}$), and $D = \{1,2\}$.

This is the reasons why a different approach is used for modeling such relations in practice - so called *global constraints* are used. A global constraint is a specialized constraint modeling a specific sub-problem. The scope of a global constraint typically includes higher number of variables. The global constraints are accompanied with a specialized and efficient algorithm for enforcing certain type of consistency.

The global constraint for the above example when different values must be assigned to variables is called *allDifferent* (Régin, 1994). Its filtering procedure for enforcing consistency is based on algorithms for computing maximum network flows (Ahuja *et al.*, 1993). This constraint is able to detect insolvability of the previously mentioned problem P. We shall define a new global constraint for planning later in the thesis.

2.8.7 Problem Modeling

Another important issue regarding the efficiency of solving of CSP is *modeling*. There are many different ways how to express a certain problem as a CSP. Models for a given problem are typically equivalent in terms of the set of solutions (it is possible to obtain a solution of one model from another model by certain transformation). However, different mod-

els for the same problem may behave differently with respect to the consistency techniques (for example arc-consistency may propagate well in some model but may not propagate in another model). Consider the following example. Let us have a CSP $P_1 = (X_1, C_1, D_1)$, where $X_1 = \{x_1, x_2, x_3\}$, $C_1 = \{x_1 = x_2; x_1 + x_2 = x_3\}$), and $D_1 = \{1,2,3\}$, and let us have a CSP $P_2 = (X_2, C_2, D_2)$, where $X_2 = \{y_1, y_2, y_3\}$, $C_2 = \{x_1 = x_2; 2x_2 = x_3\}$), and $D_2 = \{1,2,3\}$. Problems are equivalent. That is, the set of solutions of P_1 is $\{x_1 = 1, x_2 = 1, x_3 = 2\}$, the set of solutions of P_2 is $\{y_1 = 1, y_2 = 1, y_3 = 2\}$, and there is a simple transformation between the solutions $\{x_1 \leftrightarrow y_1, x_2 \leftrightarrow y_2, x_3 \leftrightarrow y_3\}$. However, arc-consistency (the generalized version for constraints of higher arities than two) for the problem P_1 infers that $D[x_1] = \{1,2\}$, $D[x_2] = \{1,2\}$, and $D[x_3] = \{2,3\}$; while arc-consistency for the problem P_2 infers that $D[y_1] = \{1\}$, $D[y_2] = \{1\}$, and $D[y_3] = \{2\}$. That is, the model P_2 allows arc-consistency to infer more information.

Different models of the same problem are often combined together to obtain strongest possible filtration by consistencies. In each such *sub-model* a consistency efficient for this sub-model is enforced. Thus a different filtering effects in each sub-model can be achieved (different sets of values are ruled out in each sub-model). Then the filtering effects are combined together using so called *channeling constraints*. A channeling constraint typically requires equivalence between sub-models for the same part of the problem. If we use the above example with sub-models P_1 and P_2, the channeling constraints would be $x_1 = y_1$, $x_2 = y_2$, and $x_3 = y_3$. For more complex problems such kind of a careful modeling based on different views of the problem can bring strong filtering effects as we shall show later for planning problems.

CHAPTER 3

CONTRIBUTIONS TO PLANNING USING PLANNING GRAPHS

We are focusing on improving of the plan extraction phase of the GraphPlan algorithm. By using concepts and techniques from *constraint programming* (Dechter, 2003) we improve the search for a plan. We are using maintaining *arc-consistency* (Mackworth, 1977; Dechter, 2003) during extraction of a plan from the planning graph to reduce the search space. Further, we proposed a special type of global reasoning which we called *projection consistency* which allows the fast and efficient reasoning about goal satisfaction within the planning graph during plan extraction. The projection consistency allows yet stronger reduction of the search space than maintaining arc-consistency. Moreover, the concept of projection consistency can be used for definition of class of sub-problems that can be solved in polynomial time.

A material presented in this chapter also appears in (Surynek, 2006), (Surynek, 2007a), (Surynek, 2007b), (Surynek, 2007c), and in (Surynek, 2007d). In (Surynek, 2006) and (Surynek, 2007a) we introduce some kind of a constraint based reasoning in *planning graphs* (Blum and Furst 1997). In (Surynek and Barták, 2007a) we study the effect of maintaining *arc-consistency* in planning graphs. The projection consistency is described in (Surynek, 2007c, 2008a; Surynek *et al.*, 2007a, 2007b). Finally, we define a tractable class based on the projection consistency in (Surynek, 2007d, 2007e, 2008b; Surynek and Barták, 2007b).

3.1 Problem of Finding Supporting Actions

We study the problem of finding a mutex-free set of actions in an action layer of the planning graph that together satisfy a certain goal (they support the goal). To distinguish between the main goal of the planning problem and goals that arise during search within plan

extraction phase we call the later *sub-goals*. The formal definition of the problem is as follows.

Definition 3.1 (PROBLEM OF FINDING SUPPORTS). Let A be a set of actions of the action layer of a given planning graph and let μA be a set of mutexes between actions from A. Next let us have a sub-goal g. A *problem of finding supports* for a sub-goal g is the task of determining a set of actions $\zeta \subseteq A$ where no two actions from ζ are mutex with respect to μA and ζ satisfies the sub-goal g, that is $g \subseteq \bigcup_{a \in \zeta} e^+(a)$. The actions from the set ζ are called *supports* for the sub-goal g in this context. □

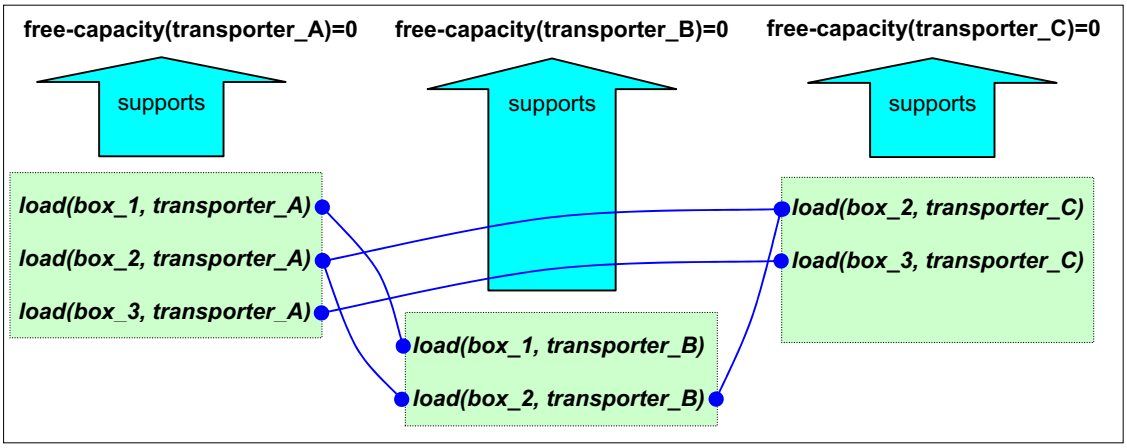


Figure 3.1. *PROBLEM OF FINDING SUPPORTS FOR A SUB-GOAL.* Example of a problem of finding supports for a sub-goal. Three transporters must be loaded. Each transporter has a capacity of one box. Mutexes are depicted as arcs between actions (mutexes are between actions loading the same box to different transporters). State variable representation is used in the figure.

An example of the problem of finding supports for a sub-goal is shown in figure 3.1. If we examine the course of execution of the GraphPlan algorithm, we can observe that typically huge numbers of sub-goals must be satisfied or proved to be unsatisfiable along the search for the global goal in the standard GraphPlan algorithm. The effectiveness of a method for solving the problem of finding supports has therefore a major impact on the performance of the planning algorithm as a whole. Unfortunately the problem of finding supports is *NP*-complete. This claim can be proved by using reduction of Boolean formula satisfaction problem to the problem of finding supports. The illustration of the reduction is also showed in figure 3.2.

Proposition 3.1 *(COMPLEXITY OF PROBLEM OF FINDING SUPPORTS). The problem of finding supports for a sub-goal in planning graph is NP-complete.* ■

Proof. Observe that the problem of finding supports is in *NP*. It is sufficient to treat sets as lists to prove this claim. Having a set of actions ζ it is possible to check whether it is a solution of the supports problem for a goal g in $O(|\zeta|(|A|+|\mu A|+|g|))$ steps. First we need check if all the actions from ζ are also from A. It takes $|\zeta||A|$ steps to check if $\zeta \subseteq A$ holds. Next we need to check if no two actions from ζ are mutex. A mutex $\{a_i, a_j\}$ where $a_i, a_j \in A$, can be checked against ζ in $2|\zeta|$ steps. For all mutexes this can be done in $2|\zeta||\mu A|$ steps. Computing of the set $\bigcup_{a\in\zeta} e^+(a)$ takes $\Sigma|\zeta|$ steps where Σ is the action size bounding constant (that is $(\forall a \in A) \quad \Sigma \geq \max(|p(a)|, |e^+(a)|, |e^-(a)|)$). Checking whether $g \subseteq \bigcup_{a\in\zeta} e^+(a)$ takes $\Sigma|\zeta||g|$ steps. Hence, the total number of steps for verifying the solution is $|\zeta||A| + 2|\zeta||\mu A| + \Sigma|\zeta||g|$ which is $O(|\zeta|(|A|+|\mu A|+|g|))$. The resulting expression is polynomial in size of the input.

Completeness with respect to *NP* class can be proved by using polynomial reduction of Boolean formula satisfaction problem (*SAT*) to problem of finding supports. Consider a Boolean formula B. It is possible to assume that the formula B is in the form of conjunction of disjunctions, that is $B = \wedge_{i=1}^{n} \vee_{j=1}^{m_i} x_j^i$, where x_j^i is a variable or a negation of a variable. For each clause $\vee_{j=1}^{m_i} x_j^i$ where $i = 1,2,\ldots,n$ we introduce a literal l_i into the constructed goal g. Next we introduce an action $a_j^i = (\varnothing, \{l_i\}, \varnothing)$ into the set of actions A for each x_j^i from the clause (the action has the only one positive effect and no preconditions and no negative effects). Actions are introduced in this way for all the clauses from B. If for some $i,k \in \{1,2,\ldots,n\}$; $j \in \{1,2,\ldots,m_i\}$; $l \in \{1,2,\ldots,m_k\}$ $x_j^i = \neg x_l^k$ or $\neg x_j^i = x_l^k$ holds we introduce a mutex $\{a_j^i, a_l^k\}$ into the set of mutexes μA (notice that it is also possible to build the set of mutexes implicitly by constructing appropriate positive and negative effects of the actions). The constructed sets $A, \mu A$ and the goal g constitute the instance of the problem of finding supports (see figure 3.2). The size of the resulting problem is $O(|B|^2)$, where $|B|$ is the number of literals appearing in B.

Having a set of actions ζ solving the constructed instance of the problem of finding supports we can construct valuation f as follows $f(x_j^i) = True$ (that is: if $x_j^i = v$ for some variable v then $f(v) = True$, if $x_j^i = \neg v$ then $f(v) = False$) for each $a_j^i \in A$. The truth values for the remaining variables in B can be selected arbitrarily. Mutexes ensure that the valuation f is a correctly defined function. Moreover, we have $f(\vee_{j=1}^{m_i} x_j^i) = True$ for $i = 1,2,\ldots,n$. Thus every clause of B is positively valued. This is implied by the fact that the goal g is satisfied by ζ. The solution of the original Boolean formula satisfaction problem is obtained from ζ in $O(|B|)$ steps. ■

There is a little hope to solve the problem of finding supporting actions effectively in general in the light of the above result. Nevertheless, the approach presented within the algorithm 2.1 where the simple uninformed backtracking is used to solve the problem is too naive.

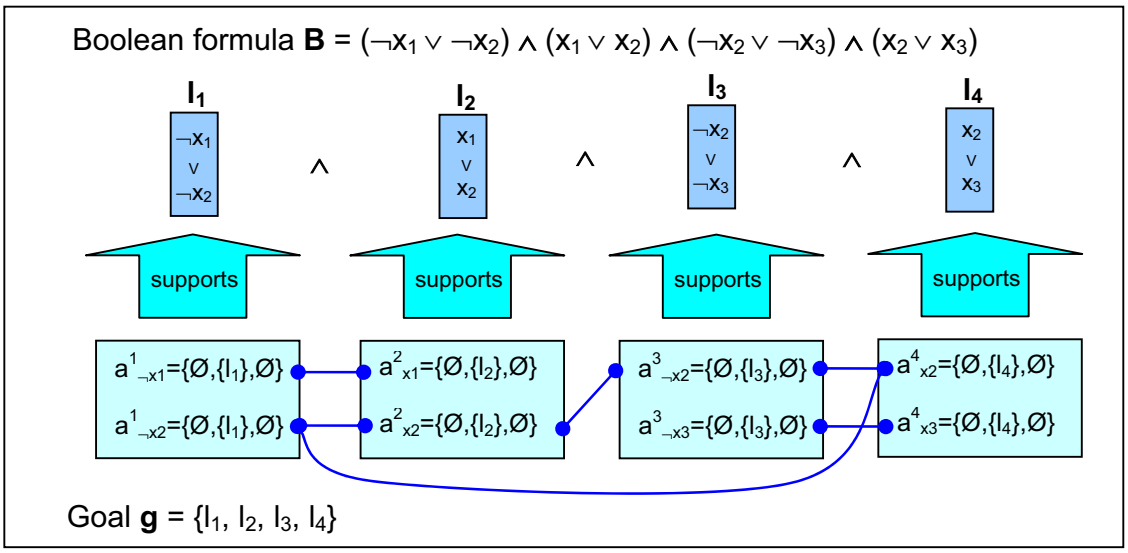


Figure 3.2. *REDUCTION OF BOOLEAN SATISFIABILITY TO THE PROBLEM OF FINDING SUPPORTS.* Illustration of the reduction of a Boolean satisfaction problem to the problem of finding supports for a sub-goal.

3.2 Basic Constraint Model: Arc-consistency

Compared to the simple uninformed backtracking used to solve the problem of finding supporting actions for a sub-goal constraint programming provides more effective tools to solve similar combinatorial problems.

We model the problem of finding supporting actions for a sub-goal as a constraint satisfaction problem. Having this formulation, we use *arc-consistency* (Mackworth, 1977) for pruning the search space during the search for supporting actions. The constraint model is built whenever a sub-goal arises in some layer of the planning graph during the plan extraction phase of the GraphPlan algorithm. Suppose that the sub-goal g' arises in course of plan extraction phase in the *i*th proposition layer of the planning graph. At this point of the plan extraction phase we build the constraint satisfaction problem expressing the problem of finding mutex-free set of actions from the *i*th action layer that support the sub-goal g'. We use two types of variables to model the problem - activity variables and support variables.

Activity variables: A Boolean variable $active(a) \in \{False, True\}$ is included in the model for every action a from the *i*th action layer of the planning graph which supports some proposition in the sub-goal g' (that is $e^+(a) \cap g' \neq \emptyset$). ●

Support variables: A variable $support(p)$ is included into the model for every atom $p \in g'$. The domain of the variable $support(p)$ consists of all the actions from the ith action layer of the planning graph which support the atom p (that is $support(p) \in \{a \mid p \in e^+(a)\}$). ●

Constraints of the model are accumulated in two clusters. The first cluster is formed by constraints between Boolean activity variables and the second cluster is represented by constraints between support variables. There is one special more complex channeling constraint between these two clusters. This special constraint controls propagation of changes between the clusters.

Activity mutex constraint: A binary constraint $activityMutex(a,b)$ forbidding assignment of the value $True$ to the pair of Boolean activity variables $active(a)$ and $active(b)$ (that is $active(a) = True \wedge active(b) = True$ is forbidden) is included into the model if and only if actions a and b are mutex in the ith action layer of the planning graph. ●

Support mutex constraint: A binary constraint $supportMutex(p,q)$ between variables $support(p)$ and $support(q)$ is refined by adding a new forbidden assignment $support(p) = a \wedge support(q) = b$ if and only if actions a and b are mutex in the ith action layer of the planning graph (more precisely, we start with the constraint $supportMutex(p,q)$ that allows arbitrary valuation of the variables of its scope respecting the domains; then for every pair of actions $a \in support(p)$ and $b \in support(q)$ that are mutex in the ith action layer of the planning graph we refine the constraint $supportMutex(p,q)$ by forbidding the new assignment $support(p) = a \wedge support(q) = b$).●

Having this model the uninformed backtracking within the plan extraction phase which solves the problem of finding supporting actions (lines 16-27 of algorithm 2.1) can be replaced by solving of the proposed constraint model. The defined model combines in fact two models. The first sub-model (cluster) consists of activity variables and activity mutex constraints, the second sub-model (cluster) consists of support variables and support mutex constraints. The cluster of activity variables is used for fast propagation after selection of an action. It provides a fast detection of actions mutually excluded with the selected action. The cluster consisting of support variables and constraints represents the main sub-model. It is richer than the activity sub-model and arc-consistency can infer more in this sub-model (the solution of activity sub-model can be derived from the solution of support sub-model; however, the opposite is not possible since there is no information about supported atoms in the activity sub-model). The drawback of the support sub-model is that it is large and in practice enforcing arc-consistency in this sub-model takes non-trivial amount of time.

To solve the proposed constraint model we use the standard chronological based backtracking (Dechter, 2003) augmented by various degrees of constraint propagation and cer-

tain type of dynamicity (the model is changed during search). The backtracking is also augmented by the standard variable and value selection heuristics (Bessière and Régin, 1996). The search proceeds by selecting an atom with the fewest number supports from the current sub-goal (that is a support variable with the smallest current domain is selected - first fail heuristic). The number of mutexes between the support variables is used for breaking ties. That is, if there are more support variables with the smallest current domain then the support variable which participates in more mutex relations is selected. For value ordering we select the value (action) that participates in the fewest number of mutex relations (succeed first heuristic). The general framework of the process of solving the proposed constraint model is shown as algorithm 3.1.

Algorithm 3.1. *SOLVING METHOD FOR BASIC CONSTRAINT MODEL OF PROBLEM OF SUPPORTS.* Top level search control of the algorithm for solving basic constraint model of the problem of finding supporting action for a sub-goal. The symbolic code uses external function *propagateBasicModel* which enforces certain level of consistency in the model. Several variants of this function are described in the following section.

function $solveBasicModel(M,\zeta)$: **set**

1: $X_{SUPPORT} \leftarrow \{support(p) \mid support(p) \in M.X\}$
2: **if** $X_{SUPPORT} = \emptyset$ **then return** ζ
3: **else**
4: $\quad x_{SUPPORT} \leftarrow heuristicallySelectVariable(X_{SUPPORT})$
5: $\quad$ **if** $M.D[x_{SUPPORT}] = \emptyset$ **then return** $\{failure\}$
6: $\quad$ **else**
7: $\quad\quad$ **while** $M.D[x_{SUPPORT}] \neq \emptyset$ **do**
8: $\quad\quad\quad a_{SUPPORT} \leftarrow heuristicallySelectValue(M.D[x_{SUPPORT}])$
9: $\quad\quad\quad M'.D[active(a_{SUPPORT})] \leftarrow M.D[active(a_{SUPPORT})] - \{False\}$
10: $\quad\quad\quad h \leftarrow \{p \mid p \in e^{+}(a_{SUPPORT})\}$
11: $\quad\quad\quad X_{DELETE} = \{support(p) \mid p \in h \wedge support(p) \in M.X\}$
12: $\quad\quad\quad M'.X \leftarrow M.X - X_{DELETE}$
13: $\quad\quad\quad C_{DELETE} \leftarrow \{c \mid c \in M.C \wedge X_c \cap X_{DELETE} \neq \emptyset\}$
14: $\quad\quad\quad M'.C \leftarrow M.C - C_{DELETE}$
15: $\quad\quad\quad (M',\zeta') \leftarrow propagateBasicModel(M',\zeta \cup \{a_{SUPPORT}\})$
16: $\quad\quad\quad$ **if** $\zeta' \neq \{failure\}$ **then**
17: $\quad\quad\quad\quad \zeta'' \leftarrow solveBasicModel(M',\zeta')$
18: $\quad\quad\quad\quad$ **if** $\zeta'' \neq \{failure\}$ **then return** ζ''
19: $\quad\quad\quad M.D[x_{SUPPORT}] \leftarrow M.D[x_{SUPPORT}] - \{a_{SUPPORT}\}$
20: $\quad\quad$ **return** $\{failure\}$

The solving algorithm is represented by the single recursive function *solveBasicModel* which gets two parameters M and ζ. The parameter M is the constraint model (constraint satisfaction problem represented as a compound structure consisting of the set of variables X, of the set of constraints C, and of the array D indexed by variables representing current variable domains; components of this compound structure are again accessed using "." - dot). The parameter ζ is the set of already selected actions into the solution of the problem. To solve the constraint model M representing the problem of finding supporting actions the function should be invoked as follows: $solveBasicModel(M, \emptyset)$.

The solving process assigns values to support variables until the set of assigned support variables satisfies the original goal. In fact we do not assign values to all the support variables. The support variables of already satisfied atoms are immediately removed from the model together with constraints containing them in their scope. If there are no support variables in the model, the problem is solved. The modification of the model is done because of its large size at the beginning. The smaller model has smaller number of constraints that must be checked for consistency.

This process is implemented by the *solveBasicModel* function as follows. First it is checked whether there are any support variables (lines 1-2). If there is no support variable the problem is solved and the parameter ζ is the solution. Otherwise a support variable $x_{SUPPORT}$ is selected (lines 4-5) and if its current domain is non-empty all the values from the current domain are tested to participate in the solution (lines 6-8; values are tested in a heuristically determined order). When a value $a_{SUPPORT}$ for the support variable $x_{SUPPORT}$ is selected the constraint model is modified by removing the support variable $x_{SUPPORT}$ and corresponding constraints (lines 8-14). The activity variable corresponding to $a_{SUPPORT}$ action is forced to be active (the *False* value is ruled out from its current domain - line 9). Then a certain level of consistency is enforced in the model (line 15). The result is a modified constraint model M' and eventually more actions selected into the solution set ζ'. If the consistency enforcing is successfully finished the solving process continues by the recursive call of the *solveBasicModel* function with modified constraint model M' and extended set of selected actions ζ' as parameters (lines 16-18).

The propagation in the model is ensured in several ways. The first aspect is the dynamicity of the model. Whenever the algorithm detects that an action must be performed to support a certain atom in the sub-goal, the constraint model is refined by deleting all the support variables that correspond to the positive effects of the action. The constraint network is also appropriately modified (all the constraints connected with the deleted variable are removed). An action a must be performed if the corresponding activity variable $active(a)$ has the singleton set $\{True\}$ as its current domain or the support variable $support(p)$ of some atom p has the singleton set $\{a\}$ as its current domain. The latter case means that the action a is the only supporting action for the atom p and thus there is the only chance to satisfy the atom p by selecting the action a to be performed.

The second aspect of constraint propagation is consistency. We maintain arc-consistency in the cluster of activity variables and in the cluster of support variables separately. The propagation between the clusters is done through the special channeling constraint (recall that arc-consistency has different effect in both clusters). We proposed three variants of constraint propagation between the clusters. The method of propagation through the channeling constraint strongly relates to the way how consistency is enforced in the model. However, the common property is that every time when the labeling step is performed (an action is selected) the consistency is enforced in the model (or more precisely, consistency is enforced in a selected part of the model). The variants of propagation are described in the following section. For enforcing arc-consistency in the model the algorithm AC-3 is used (see section 2.9.4).

3.2.1 Variants of Constraint Propagation

We describe three variants of propagation scheme for the algorithm for solving the basic constraint model of the problem of finding supporting actions for a sub-goal. The individual variants represent different levels of consistencies enforced in the constraint model. The variants are named: *variant A*, *variant B*, and *variant C*. The *variant A* represents the weakest level of consistency; on the other hand this variant propagates quickly. The *variant C* represents the strongest level of consistency but it is computationally most expensive. The *variant B* represents a compromise between *variant A* and *variant C*.

The individual propagation schemes can be integrated with the top control solving algorithm (algorithm 3.1) by appropriate definition of the external function *propagate-BasicModel* (line 15).

Propagation scheme of *variant A*: When a supporting action is selected to satisfy an atom in the sub-goal, the corresponding activity variable is set to the value *True*. Then arc-consistency is enforced in the cluster of activity variables. The next step consists of propagation of the changes in the cluster of activity variables into the cluster of support variables through the channeling constraint. The channeling constraint is defined as follows in this variant. If an activity variable is definitely *False* (its current domain is singleton $\{False\}$), then the corresponding action is removed from current domains of all the supporting variables. Observe that this variant cannot detect that an activity variable is definitely *True* (its current domain is singleton $\{True\}$). That is why arc-consistency propagates weakly through the activity mutex constraints.

This variant is functionally equivalent to forward checking (Dechter, 2003) in the cluster of supporting variables (however this version is faster thanks to simplicity of the activity cluster).

The whole propagation variant is formally described here as algorithm 3.2. The algorithm is represented by the single function *propagateBasicModel-VariantA* which gets the constraint model M and the set of already selected actions into the solution ζ as parameters. The function returns a pair consisting of a modified constraint model after enforcing consistency and of the set of actions selected into the solution (in this case the set of selected actions is the same as the input parameter; in other variants it is extended). Arc-consistency is enforced within the cluster of activity variables which is denoted as $M|_{X_{ACTIVITY}}$ (the constraint model restricted to the set of activity variables). After enforcing the consistency (lines 1-2) the changes made in the cluster of activity variables are propagated into the cluster of support variables (lines 3-8). ●

Algorithm 3.2. *CONSTRAINT PROPAGATION - VARIANT A*. This algorithm represents a variant of the external function *propagateBasicModel* for the algorithm 3.1.

function $propagateBasicModel\text{-}VariantA(M,\zeta)$: **pair**
1: $X_{ACTIVITY} \leftarrow \{active(a) \mid active(a) \in M.X\}$
2: $M|_{X_{ACTIVITY}} \leftarrow enforceArcConsistency\text{-}AC\text{-}3(M|_{X_{ACTIVITY}})$
3: $A_{REMOVE} \leftarrow \{active(a) \mid active(a) \in M.X \wedge M.D[active(a)] = \{False\}\}$
4: **for each** $active(a_{REMOVE}) \in A_{REMOVE}$ **do**
5: | **for each** $support(p) \in M.X$ **such that** $a_{REMOVE} \in M.D[support(p)]$ **do**
6: | | $D[support(p)] \leftarrow D[support(p)] - \{a_{REMOVE}\}$
7: | | **if** $D[support(p)] = \emptyset$ **then return** $(M, failure)$
8: **return** (M,ζ)

Propagation scheme of *variant B*: In this propagation scheme we proceed similarly as in the *variant A*. When a supporting action is selected to satisfy some atom in the goal the corresponding activity variable is set to the value *True*. Then arc-consistency is enforced in the cluster of activity variables and changes are propagated into the cluster of support variables. This propagation is done in the same way as in the *variant A*. In addition to the *variant A*, changes in the cluster of support variables are propagated back to the cluster of activity variables. It is done in the following way. When a support variable has a singleton set as its current domain (the proposition has the only support) the corresponding activity variable is set to the value *True* and arc-consistency is enforced again in the cluster of activity variables. The process is repeated until changes are made.

The symbolic code of this propagation variant is listed algorithm 3.3. The algorithm is again represented by the single function *propagateBasicModel-VariantB* which has the same interface as the previous variant of propagation function (input parameter and return values are the same). The function consists of a loop (lines 2-22) which continues until the constraint model is changing. Arc-consistency is enforced in the cluster of activity variables (lines 12-13). Then changes made in the previous execution of the loop in the cluster of

support variables are propagated into the cluster of activity variables - the constraint model is modified (lines 3-11). Next, the changes made in the cluster of activity variables are propagated into the cluster of support variables (lines 14-21). ●

Algorithm 3.3. *CONSTRAINT PROPAGATION - VARIANT B*. This algorithm represents a variant of the external function *propagateBasicModel* for the algorithm 3.1.

function *propagateBasicModel-VariantB*(M,ζ): **pair**

1: $A_{SELECT} \leftarrow \emptyset$
2: **do**
3: | **for each** $active(a_{SELECT}) \in A_{SELECT}$ **do**
4: | | $M.D[active(a_{SELECT})] \leftarrow M.D[active(a_{SELECT})] - \{False\}$
5: | | $h \leftarrow \{p \mid p \in e^{+}(a_{SELECT})\}$
6: | | $X_{DELETE} = \{support(p) \mid p \in h \wedge support(p) \in M.X\}$
7: | | $M.X \leftarrow M.X - X_{DELETE}$
8: | | $C_{DELETE} \leftarrow \{c \mid c \in M.C \wedge X_c \cap X_{DELETE} \neq \emptyset\}$
9: | | $M.C \leftarrow M.C - C_{DELETE}$
10: | | $A_{SELECT} \leftarrow A_{SELECT} - \{active(a_{SELECT})\}$
11: | | $\zeta \leftarrow \zeta \cup \{a_{SELECT}\}$
12: | $X_{ACTIVITY} \leftarrow \{active(a) \mid active(a) \in M.X\}$
13: | $M|_{X_{ACTIVITY}} \leftarrow enforceArcConsistency\text{-}AC\text{-}3(M|_{X_{ACTIVITY}})$
14: | $A_{REMOVE} \leftarrow \{active(a) \mid active(a) \in M.X \wedge M.D[active(a)] = \{False\}\}$
15: | **for each** $active(a_{REMOVE}) \in A_{REMOVE}$ **do**
16: | | **for each** $support(p) \in M.X$ **such that** $a_{REMOVE} \in M.D[support(p)]$ **do**
17: | | | $D[support(p)] \leftarrow D[support(p)] - \{a_{REMOVE}\}$
18: | | | **if** $D[support(p)] = \emptyset$ **then return** $(M, failure)$
19: | | | **else**
20: | | | | **if** $D[support(p)] = \{a\}$ **then**
21: | | | | | $A_{SELECT} \leftarrow A_{SELECT} \cup \{active(a)\}$
22: **while** $A_{SELECT} \neq \emptyset$
23: **return** (M,ζ)

Propagation scheme of *variant C*: This variant further evolves the previous variant. Now consistency is enforced in both clusters. After selecting the action to support the given atom a corresponding activity variable is set to the value *True* and arc-consistency is enforced in the cluster of activity variables. Then changes are propagated into the cluster of support variables where arc-consistency is enforced too. The last step of the iteration consists of propagation of changes from the cluster of support variables into the cluster of activity variables. Propagation in both directions between variable clusters through channeling con-

straint is done in the same way as in previous variants. The whole process is again repeated until the model in changing.

The symbolic code of this propagation variant is listed here as algorithm 3.4. The symbolic code of this propagation variant is similar to the *variant B*. Changes are again propagated from the cluster of support variables into the cluster of activity variables (lines 3-11) and in the opposite direction from the cluster of activity variables into the cluster of support variables (lines 15-18). The difference from the *variant B* is that arc-consistency is now enforced both in the cluster of activity variables (lines 13-14) and in the cluster of support variables (lines 19-20). ●

Algorithm 3.4. *CONSTRAINT PROPAGATION - VARIANT C*. This algorithm represents a variant of the external function *propagateBasicModel* for the algorithm 3.1.

function *propagateBasicModel-VariantC*(M,ζ) : **pair**
1: $A_{SELECT} \leftarrow \emptyset$
2: **do**
3: | **for each** $active(a_{SELECT}) \in A_{SELECT}$ **do**
4: | | $M.D[active(a_{SELECT})] \leftarrow M.D[active(a_{SELECT})] - \{False\}$
5: | | $h \leftarrow \{p \mid p \in e^{+}(a_{SELECT})\}$
6: | | $X_{DELETE} = \{support(p) \mid p \in h \wedge support(p) \in M.X\}$
7: | | $M.X \leftarrow M.X - X_{DELETE}$
8: | | $C_{DELETE} \leftarrow \{c \mid c \in M.C \wedge X_c \cap X_{DELETE} \neq \emptyset\}$
9: | | $M.C \leftarrow M.C - C_{DELETE}$
10: | | $A_{SELECT} \leftarrow A_{SELECT} - \{active(a_{SELECT})\}$
11: | | $\zeta \leftarrow \zeta \cup \{a_{SELECT}\}$
12: | $X_{ACTIVITY} \leftarrow \{active(a) \mid active(a) \in M.X\}$
13: | $M|_{X_{ACTIVITY}} \leftarrow enforceArcConsistency\text{-}AC\text{-}3(M|_{X_{ACTIVITY}})$
14: | $A_{REMOVE} \leftarrow \{active(a) \mid active(a) \in M.X \wedge M.D[active(a)] = \{False\}\}$
15: | **for each** $active(a_{REMOVE}) \in A_{REMOVE}$ **do**
16: | | **for each** $support(p) \in M.X$ **such that** $a_{REMOVE} \in M.D[support(p)]$ **do**
17: | | | $D[support(p)] \leftarrow D[support(p)] - \{a_{REMOVE}\}$
18: | | | **if** $D[support(p)] = \emptyset$ **then return** $(M, failure)$
19: | $X_{SUPPORT} \leftarrow \{support(p) \mid support(p) \in M.X\}$
20: | $M|_{X_{SUPPORT}} \leftarrow enforceArcConsistency\text{-}AC\text{-}3(M|_{X_{SUPPORT}})$
21: | **for each** $support(p) \in M.X$ **such that** $M.D[support(p)] = \{a\}$ **do**
22: | | $A_{SELECT} \leftarrow A_{SELECT} \cup \{active(a)\}$
23: **while** $A_{SELECT} \neq \emptyset$
24: **return** (M,ζ)

The constraint model with maintaining consistency provide stronger search space pruning than the approach used within the standard GraphPlan's plan extraction phase. The question is which variant performs best and what type of consistency is better. Experiments showed that *variant C* is the best choice on problems with higher number of interacting objects and with high action parallelism. However on problems with low object interaction the simple *variant A* is the best. Let us note that the cluster of action variables provides faster constraint propagation compared to the cluster of support variables since it is structurally simpler.

The space complexity of the constraint model corresponds to the space complexity of the layer in which the constraint model is build. This observation is summarized in the following proposition.

Proposition 3.2 *(SPACE COMPLEXITY OF BASIC CONSTRAINT MODEL). The worst case estimation on the space required by the basic constraint model for the problem of finding supporting actions in the* i*th action layer* A_i *for a sub-goal* g *which arise in the* i*th proposition layer* P_i *of the planning graph is* $O(|g||A_i|+|\mu A_i|)$. ■

Proof. The constraint model consists of $|A_i|$ activity variables with domains of size 2 and of $|g|$ support variables with domains of sizes at most $|A_i|$. Next we need a space of $|\mu A_i|$ to store the constraints in the model. Together we need the space of $2|A_i|+|g||A_i|+|\mu A_i|$ which is $O(|g||A_i|+|\mu A_i|)$. ■

The worst case time complexity of solving the problem of finding supports is exponential unless $P = NP$. However, the interesting information with respect to the analysis is time complexity of a single propagation step. The results are summarized in the following propositions.

Proposition 3.3 *(TIME COMPLEXITY OF A SINGLE PROPAGATION STEP OF VARIANT A). The worst case time complexity of a single propagation step of the propagation variant A is* $O(|A_i|^2+|g||A_i|)$ *supposed that the constraint model was built for problem of finding supports in the* i*th action and proposition layer of the planning graph and for the sub-goal* g. ■

Proof. Enforcing arc-consistency by AC-3 algorithm within the cluster of activity variables takes $O(|A_i|^2)$ steps (according to proposition 2.6 we get $O(|A_i|^2 2^3)$ which is $O(|A_i|^2)$). The propagation of changes made in the cluster of activity variables to the cluster of support variables takes $|g||A_i|$ steps since at most $|g||A_i|$ values are removed from the current variable domains in the cluster of support variables. The worst case number of steps is $O(|A_i|^2+|g||A_i|)$ in total. ■

Proposition 3.4 *(TIME COMPLEXITY OF A SINGLE PROPAGATION STEP OF VARIANT B). The worst case time complexity of a single propagation step of the propagation variant B is* $O(|A_i|^3 + |g||A_i|)$ *supposed that the constraint model was built for problem of supports in the* i*th action and proposition layer of the planning graph and for the sub-goal* g. ■

Proof. We already know that enforcing arc-consistency by AC-3 algorithm in the cluster of activity variables takes $O(|A_i|^2)$ steps. This arc-consistency enforcing must be performed $|A_i|$ times in the worst case since each iteration of the main control loop rules out from consideration at least one activity variable (activity variable is selected). Amortized number of steps over all the iterations of the main loop consumed by propagating changes between the variable clusters is $|g||A_i|$. In total we have $O(|A_i|^3 + |g||A_i|)$ steps required by the propagation of the *variant B*. ■

Proposition 3.5 *(TIME COMPLEXITY OF A SINGLE PROPAGATION STEP OF VARIANT C). The worst case time complexity of a single propagation step of the propagation variant B is* $O(|g|^2 |A_i|^3)$ *supposed that the constraint model was built for problem of finding supports in the* i*th action and proposition layer of the planning graph and for the sub-goal* g. ■

Proof. The number of steps required by enforcing arc-consistency by the algorithm AC-3 in the cluster of activity variables amortized over all the iterations of the main control loop is $O(|A_i|^3)$ in the worst case. The number of steps required by the propagation between activity variable and support variable clusters amortized over all the iterations of the main control loop is $|g||A_i|$. The number of steps amortized over all the iterations of the main control loop required by enforcing arc-consistency by the AC-3 algorithm in the cluster of support variables is $|g|^2 |A_i|^3$ (we have at most $|g|^2$ constraints and the size of the domains is bounded by $|A_i|$). Altogether we need $O(|A_i|^3) + O(|g||A_i|) + O(|g|^2 |A_i|^3)$ which is $O(|g|^2 |A_i|^3)$. ■

Propagation variant	**Variant A**	**Variant B**	**Variant C**
Time complexity (worst case)	$O(\|A_i\|^2 + \|g\|\|A_i\|)$	$O(\|A_i\|^3 + \|g\|\|A_i\|)$	$O(\|g\|^2 \|A_i\|^3)$
Space complexity (worst case)	$O(\|g\|\|A_i\| + \|\mu A_i\|)$		

Table 3.1. *COMPLEXITY OF PROPAGATION IN BASIC CONSTRAINT MODEL.* Summary of asymptotic worst case time and space complexities of the individual variants of propagations in the basic constraint model.

The above complexity results for enforcing arc-consistency in the constraint model of the problem of finding supporting actions are summarized in table 3.1.

Notice that worst case time complexities of the individual variants of propagation differ significantly. However, the time consuming propagation *variant C* may reduce the number of steps of the top control solving algorithm more than for example fast propagating *variant A*. For uncovering the behavior of the process of solving constraint models connected with various constraint propagation variants we used an experimental evaluation.

3.2.2 Experimental Evaluation

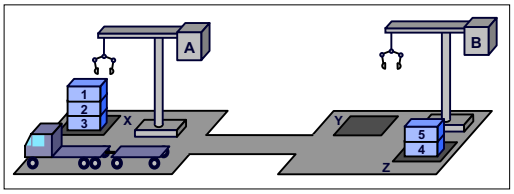
We performed a set of experiments to evaluate the contribution of the proposed constraint model and arc-consistency maintaining framework for solving the problems of finding supporting actions. Our experiments are targeted on comparison of the basic version of the GraphPlan algorithm with the enhanced versions of the GraphPlan algorithm which use the constraint model and a variant of solving technique (*variant A*, *variant B*, and *variant C*).

We implemented all the tested algorithms. This ensures that the all tested techniques use common style of programming and common data structures for the same purposes (the possible advantage of using more advanced data structure for the same task is therefore eliminated). All the tested algorithms were implemented in the C++ language (Stroustrup, 1986) and were compiled under identical conditions using gcc compiler version 3.4.3 (GNU Project, 2008) with options providing maximum optimization for the target testing machine (-O3 -mtune=opteron). The tests were run on a machine with two AMD Opteron 242 processors (1600 MHz) with 1GB of memory under Mandriva Linux 10.2 (Mandriva, 2008).

We used standard variable and value selection heuristics as mentioned above. Specifically an atom with the smallest number of supporting actions is always selected as first to be satisfied (in both the enhanced GraphPlan implementation with maintaining arc-consistency in the constraint model as well as in the standard GraphPlan implementation). Then supporting action are tried starting with the action that is least constrained (however, this value ordering was observed to has almost no effect on the performance).

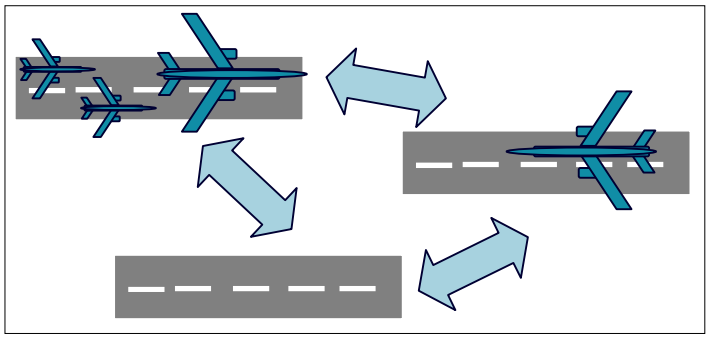
There is also another important implementation issue concerning nogood recording. We used unrestricted nogood recording. We also used state variable representation for planning problems.

We used three types of planning environments - dock worker robots environment (Ghallab *et al.*, 2004), refueling planes environment (which is original) and towers of Hanoi planning environment. Several instances of planning problems of various difficulties from each proposed planning environment were used.



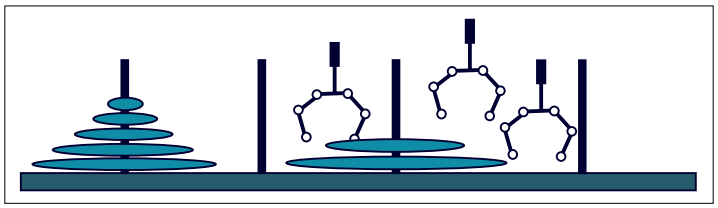
Dock Worker Robots planning environment. This planning domain consists of a traffic network, transportation trucks and of cranes (Ghallab *et al.*, 2004). Each transportation truck has a certain capacity of packages and can move arbitrarily within the traffic network. There are two types of places within the traffic network called locations and sites. A location is an ordinary place which represents a node in the traffic network. A site is a special place where packages can be loaded and unloaded to and from the transportation truck. Each site has a certain number of cranes and a certain number of piles of packages (packages in a pile behave like a stack - LIFO). Each crane can load and unload a package to and from a transportation truck. Typically, not all the piles within a site are reachable by a single crane so the cooperation among cranes on the site is necessary.

The task within this planning domain is usually to transport some packages from one site to another site and to stack them on piles in the right order. ●



Refueling Planes planning environment. Consider that we need to plan how to refuel a fleet of planes in order to get them to the far destination. For simplicity we have several airports and several planes with certain fuel capacities. Planes can travel between the airports. A plane consumes certain amount of fuel to travel a unit of distance. Some extra fuel is also necessary for landing and taking-off. Each airport has an unlimited source of fuel and planes can refuel at the airport. The important ability of planes is to transfer fuel from one plane to another plane in-flight.

The task is typically to get a fleet of planes from one airport to some distant one. The task is especially interesting when planes need an intermediate landing on some middle airport or in-flight refueling. ●



Towers of Hanoi planning environment. This planning environment is a generalization of the well known puzzle. The original game consists of three pegs and a number of discs of different sizes stacked on pegs. It is possible to move a disc on the top of one peg to another

peg in each turn. The condition that a smaller disc is always put on larger disc must be preserved throughout the game. Our generalization is that we use arbitrary number of pegs and we allow moving more than one disc in a single turn (we can pick up for example two top discs by two hands and then place them in a different order than they were picked).

The original game starts with all disc stacked on the first peg. However, we allow arbitrary configuration (satisfying the condition on disc sizes) as the starting point in our generalization. Originally, the objective is to move all discs to the last third peg. Again we allow arbitrary valid configuration as a goal. ●

Various difficulties of individual planning problems were established by using various numbers of objects appearing in the planning environments and by encoding planning tasks of different difficulties into the problems (that is we encoded tasks requiring various numbers of steps to be finished). All the problems used in our experimentation were solvable (that is, the algorithm terminated with the answer that solution exists and retuned one solution).

Along the execution of experimental tests, we collected variety of statistical data characterizing performance of the individual algorithmic techniques. The list of statistical characteristics collected is summarized in table 3.2.

Statistical data collected during tests	
Number of ***actions*** considered	Number of actions considered during plan extraction phase. The considered action is an action that was tried to be included in the resulting plan.
Number of ***backtracks***	Number of backtracks that happened during plan extraction phases.
Number on ***constraint checks***	Number checks of constraints made along the search for solution. A check of a constraint is checking whether the constraint is satisfied for a given assignment of variables of its scope.
Number of ***sub-goals***	Number of sub-goals that must be resolved during plan extraction phases. In other words, this is the number of problems of finding supporting actions that must be solved along the search for a plan.
Number of ***mutex checks***	Number of checks whether certain two actions or atoms are in the mutex relation.
Number of ***nogoods*** recorded	Number of nogoods recorded until the solution was found.
Planning graph ***building time***	Overall time spent by building and extending of the planning graph.
Plan ***extraction time***	Overall time spent in plan extraction phases.

Table 3.2. *STATISTICAL CHARACTERISTICS COLLECTED DURING EXPERIMENTAL EVALUATION.* List of statistical characteristics collected in each execution of empirical test for the individual algorithmic technique.

The most relevant values obtained from measurements are plan extraction time, number of constraint checks (it is a check whether the constraint is satisfied for a given assignment of values to variables of its scope), and number of backtracks. They represent the effectiveness of the individual algorithmic technique for solving the given problem.

Resulting solution concurrent plan characteristics are shown in tables 3.3, 3.4, and 3.5. Problems are identified by numbers for reference to the attached medium (the numbering is not continuous since some problems have very similar characteristics and therefore only one representative among similar problems were selected for presentation). The number of actions in the resulting concurrent plan and the length of concurrent plan are shown. While the lengths of concurrent plans are the same for all the methods, the number of actions may differ due to the ordering heuristics (however the difference is not significant). The tables show the number of actions of the best performing method we implemented (tractable class method - see section 3.4).

Resulting concurrent plan lengths for Dock Worker Robots (dwr) problems																
Problem number	01	02	03	04	05	07	16	17	20	21	22	23	24	25	26	27
Concurrent plan length	6	6	2	3	14	16	18	20	20	13	11	12	13	13	11	10
Number of actions	10	8	4	4	34	40	44	42	32	24	26	28	30	30	26	28

Table 3.3. *CHARACTERISTICS OF SOLUTION CONCURRENT PLANS FOR DOCK WORKER ROBOTS PROBLEMS.* Concurrent plan lengths and number of actions in concurrent plans for Dock Worker Robots problems used in experimental evaluation.

Resulting concurrent plan lengths for Towers of Hanoi (han) problems																
Problem number	01	02	03	04	07	08	09	10	11	12	13	14	15	16	17	18
Concurrent plan length	6	14	30	10	14	20	16	20	12	16	12	12	6	10	6	6
Number of actions	6	14	30	12	20	26	20	24	16	24	20	20	14	18	10	14

Table 3.4. *CHARACTERISTICS OF SOLUTION CONCURRENT PLANS FOR TOWERS OF HANOI PROBLEMS.* Concurrent plan lengths and number of actions in concurrent plans for Towers of Hanoi problems used in experimental evaluation.

The comparison of the overall solving time of the standard GraphPlan algorithm and the enhanced versions based on maintaining arc-consistency is shown in figure 3.3. The standard GraphPlan is compared with arc-consistency propagation variants *A*, *B*, and *C*. Figure 3.3 shows the time using logarithmic scale for time. Problems are ordered along the horizontal line. Each problem is identified by a prefix ("dwr" for Dock Worker Robots

planning environment, “han” for Towers of Hanoi planning environment, and “pln” for Refueling Planes planning environment) followed by the number of the problem. Problems are listed along the horizontal axis in the ascending order according to the solving time using *variant A* propagation scheme (this ordering allows to depict deviation of the standard GraphPlan and the *variant C* propagation scheme). The time limit of 1 hour is used.

Resulting concurrent plan lengths for Refueling Planes (pln) problems																
Problem number	01	04	05	06	10	11	13	14	15	16	17	19	20	21	22	23
Concurrent plan length	5	5	6	9	10	10	10	8	8	5	8	8	9	10	9	13
Number of actions	9	9	14	14	15	14	14	12	16	13	12	16	17	18	14	21

Table 3.5. *CHARACTERISTICS OF SOLUTION CONCURRENT PLANS FOR REFUELING PLANES PROBLEMS.* Concurrent plan lengths and number of actions in concurrent plans for Refueling Planes problems used in experimental evaluation.

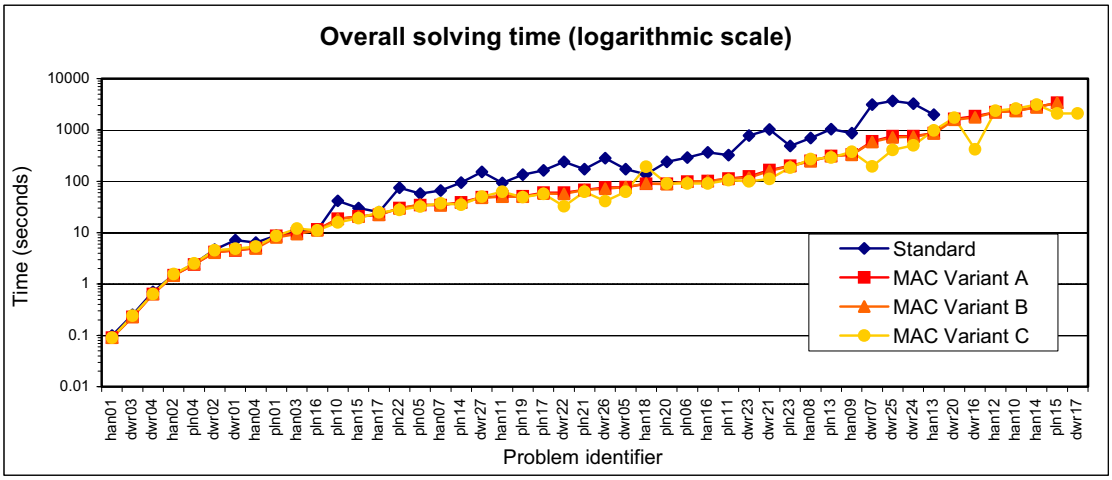


Figure 3.3. *COMPARISON OF OVERALL SOLVING TIMES (LOGARITHMIC SCALE) - (STD, VARA, VARB, VARC).* Comparison of the overall solving time of the standard GraphPlan algorithm and enhanced versions which use maintaining arc-consistency for solving the problem of finding supports (standard version and variants *A*,*B*, and *C* of the propagation scheme are compared). Problems on the horizontal axis are listed in the ascending order according to the time consumed by *variant A*. Time limit of 1 hour for each problem is used.

The graph shows that GraphPlan enhanced by any of the variants of propagation for maintaining arc-consistency in the constraint model of the problem of finding supporting actions outperforms the standard version in terms of overall problem solving time (for example the improvement is up to 1600% when we compare standard GraphPlan and the

variant C on the problem dwr07). Generally we can conclude that the improvement is better towards harder problems (harder problems are on the right). Notice that we improved only the plan extraction phase. Phase of building planning graph remains the same, so the more time is spent in plan extraction phase the better is the improvement.

If we compare the individual propagation variants, it is possible to conclude that *variant A* and *variant B* are almost the same in terms of overall solving time. The *variant B* seems to be slightly faster than the *variant A* but the difference is not significant. The more interesting difference is between the variants *A* and *B* and the *variant C*. The *variant C* is significantly faster on number of evaluated problems (namely on the problem dwr16 the *variant C* is approximate 400% faster than the variants *A* and *B*). Moreover, only the *variant C* solved all the problems within the time limit of 1 hour for each problem (all the algorithms except *variant C* failed to solve the problem dwr17; the standard GraphPlan failed to solve 7 hardest problems). However, the *variant C* is not always the fastest. Its performance is worse than that of variants *A* and *B* on certain problems (for example on the problem han14). Moreover, the variant C was outperformed by standard GraphPlan algorithm on the problem han18. This is caused by the fact that on certain types of planning problems (such as that of Towers of Hanoi) the complex propagation scheme of the *variant C* represents an overhead.

The comparison of the time spent in plan extraction phases (that is we do not account time spent by building planning graph) is shown in figure 3.4.

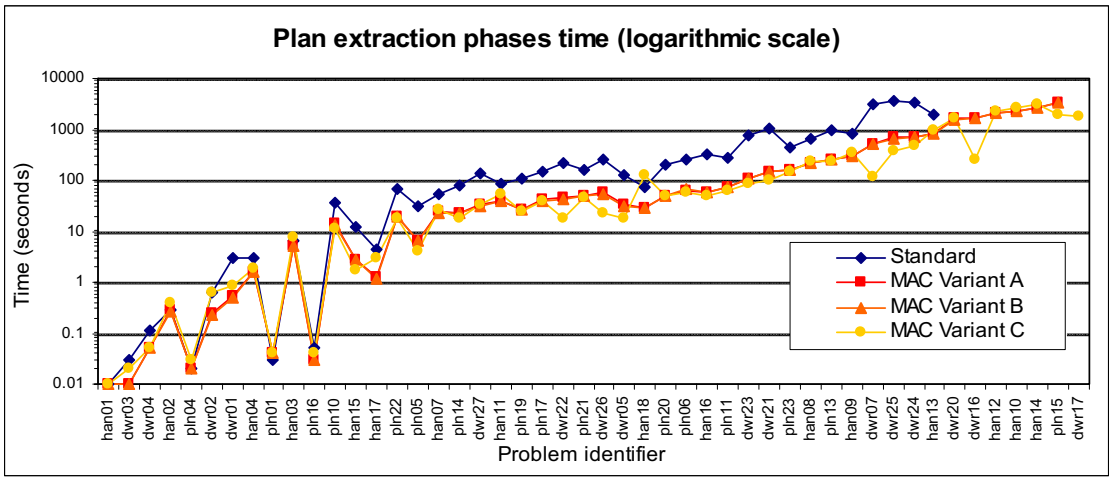


Figure 3.4. *COMPARISON OF PLAN EXTRACTION PHASES TIMES (LOGARITHMIC SCALE) - (STD, VARA, VARB, VARC).* Comparison of the plan extraction time of the standard GraphPlan algorithm and enhanced versions which use maintaining arc-consistency for solving the problem of finding supports (standard version and variants *A*,*B*, and *C* of the propagation scheme are compared). Problems on the horizontal axis are listed in the ascending order according to the overall solving time consumed by *variant A* (same ordering as in figure 3.3). Time limit of 1 hour for each problem is used again.

The ordering of problems along the horizontal axis in figure 3.4 is the same as in figure 3.4. So, the portion of time spent by building planning graphs and in plan extraction phase can be observed. The differences among individual methods are more expressed since the graph building time (which is the same for all the methods) is ruled out. The logarithmic scale is used again for the time axis.

The comparison in terms of number of constraint checks of the standard GraphPlan and the enhanced versions based on maintaining arc-consistency is shown in figure 3.5. The constraint check is checking whether a certain constraint is satisfied for a tuple of values. The ordering of the problems is again the same as in figure 3.3 and again the logarithmic scale is used.

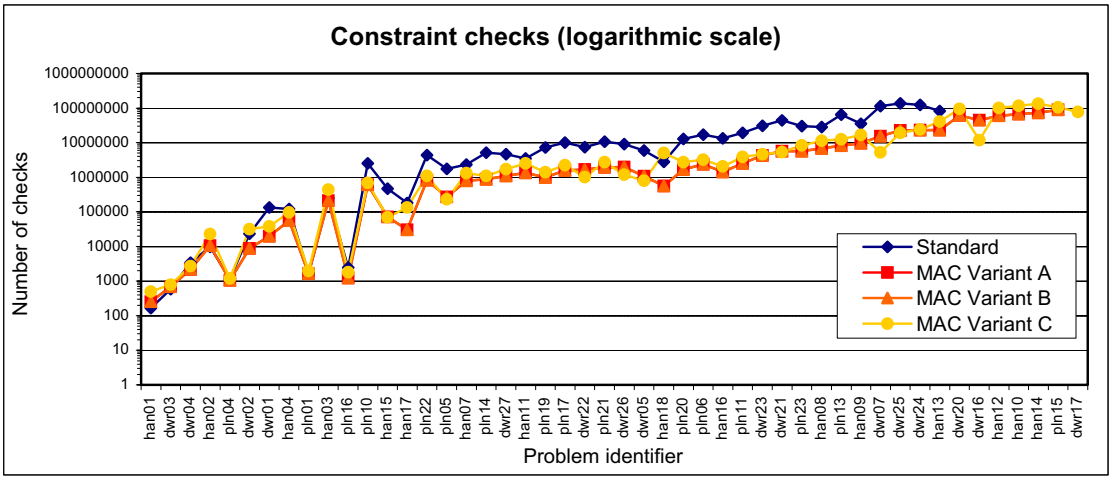


Figure 3.5. *COMPARISON OF NUMBER OF CONSTRAINT CHECKS - (STD, VARA, VARB, VARC).* Comparison of the standard GraphPlan and enhanced variants based on constraint model with maintaining arc-consistency for solving the problems of finding supports in terms of number of constraint checks. Standard GraphPlan and propagation schemes of variants *A*, *B*, and *C* are compared. The ordering of problems along the horizontal axis is the same as in figure 3.3.

The results in the graph in figure 3.5 show that the standard GraphPlan has the highest number of constraint checks (in this case it is the number of checking whether two actions are mutex). The enhanced variants of GraphPlan using propagation schemes *A* and *B* have very similar number of constraint checks. The *variant C* has the number of constraint checks slightly higher than variants *A* and *B*. This is partially expectable since the *variant C* uses the most complicated propagation scheme.

The comparison of the number of backtracks of the tested methods is shown in figure 3.6. The ordering of problems along the horizontal axis is again the same as in figure 3.3 and again the logarithmic scale is used. So, the correlation between other measured characteristics may be observed. The number of backtracks correlates quite well with the time spent in the plan extraction phase. Although not perfectly.

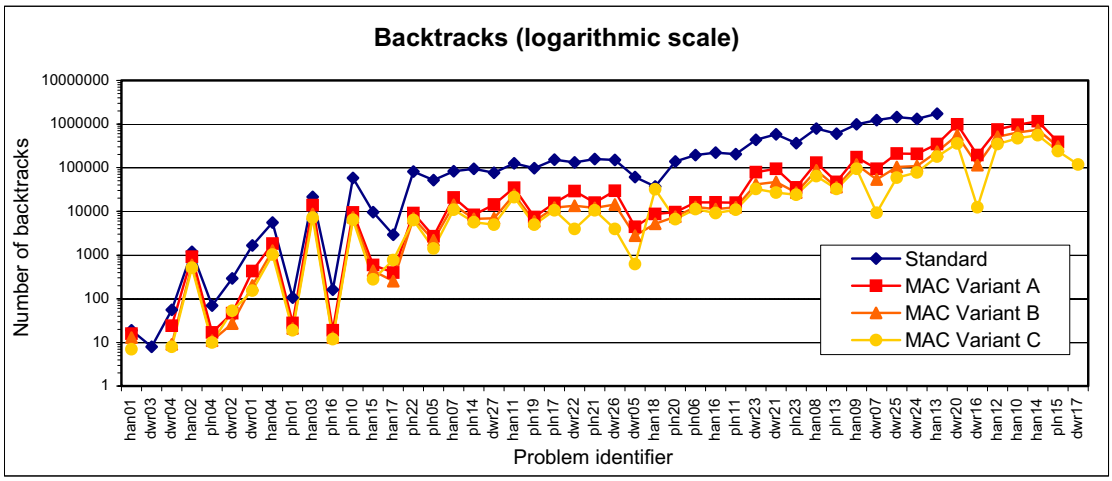


Figure 3.6. *COMPARISON OF NUMBER OF BACKTRACKS - (STD, VARA, VARB, VARC).* Comparison of the standard GraphPlan and enhanced variants based on constraint model with maintaining arc-consistency for solving the problems of finding supports in terms of number backtracks. Standard GraphPlan and propagation schemes of variants *A*, *B*, and *C* are compared. The ordering of problems along the horizontal axis is the same as in figure 3.3.

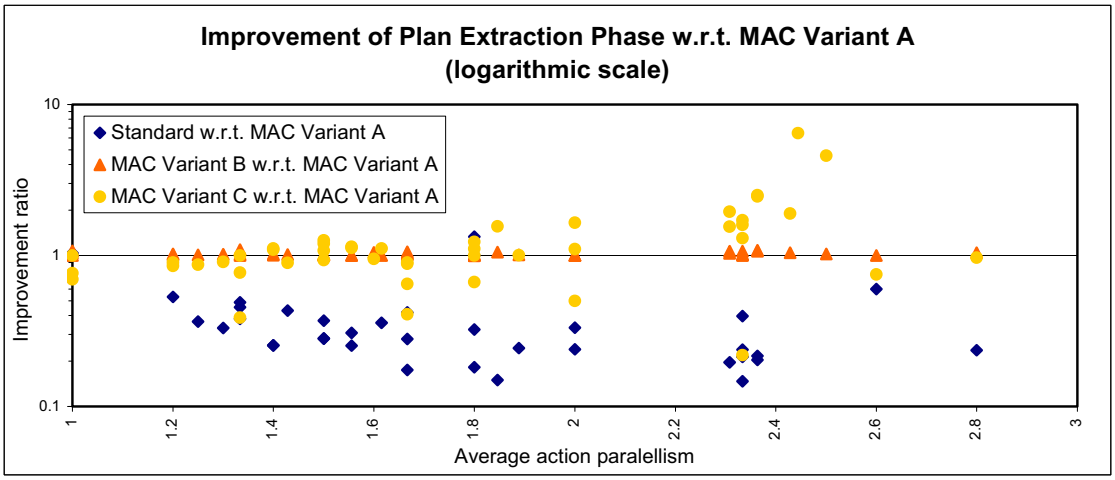


Figure 3.7. *COMPARISON OF IMPROVEMENTS WITH RESPECT TO MAC VARIANT A - (STD, VARB, VARC).* Comparison of the standard GraphPlan and enhanced variants based on constraint model with maintaining arc-consistency for solving the problems of finding supports in terms of improvement ratio of the plan extraction phase depending on the average action parallelism (number of actions in the plan divided by the length of the resulting concurrent plan). Improvements are computed with respect to the with respect to the *variant A* (which has the ratio 1).

Figure 3.6 shows that in terms of the number of backtracks the standard GraphPlan performs worst. The enhancements that use the propagation variants *A* and *B* have the similar numbers of backtracks for each problem. The best method in terms of the number of

backtracks is propagation of the *variant C*. This was expected since quite complex reasoning is done at each decision step which reduces the number of decision steps as a result.

The graph in figure 3.7 is targeted on discovering the property which relates to the achieved improvements with respect to the maintaining arc-consistency in *variant A*. The hypothesis is that higher action parallelism allows better improvements by using the constraint models and maintaining arc-consistency for solving the problem of finding supporting actions (intuitively said the action parallelism is the number of actions that can be performed simultaneously). The ordering of problems along the horizontal axis in figure 3.7 is therefore according to the number of actions in the resulting concurrent plan divided by the length of the concurrent plan. Let this value define an *action parallelism*. Although the results are not entirely convincing it seem that the higher action parallelism allows more significant improvement by solving the problem of finding supporting actions using the proposed constraint models with maintaining arc-consistency. The propagation *variant C* satisfies this statement most visibly.

3.2.3 Overall Analysis of Results

The experiments showed that maintaining arc-consistency brings a significant improvement in the number of backtracks when solving the problem of finding supporting actions for a sub-goal. There is also a significant improvement of the time of extraction phase as well as in overall solving time in comparison with standard GraphPlan.

We can conclude that the most complex propagation *variant C* performs generally as best. The gain from the *variant C* is more significant on hard problems. However, the *variant C* of propagation is not always the best choice. For example on some problems from the Hanoi Towers planning environment sometimes the *variant C* was worse than the standard version of the GraphPlan algorithm in terms of time necessary for plan extraction phase.

We also found that improvements can be achieved for problems with higher action parallelism. This is expectable since in such a case the problem of finding supporting actions is non-trivial and better reasoning about the problem pays-off. On the other hand when problems do not have high action parallelism the constraint model with maintaining arc-consistency does not bring any significant improvement. This is caused by the fact that problems of finding supporting actions are simple in these cases and maintaining arc-consistency to solve these easy problems represents an overhead in such case.

3.2.4 Discussion and Related Works

Regarding experimental evaluation there may be objection why we did not compared our planner with today's state-of-the-art planners on standard benchmark problems used in the

international planning competitions such as *IPC* (Gerevini *et al.*, 2006). The answer is as follows. First, our goal was not to implement a competitive planner for some kind of a competition. We are rather focusing on understanding the structure of planning problems and on utilizing this knowledge to improve the solving process. Second, our experimental evaluation is targeted on comparing ideas and algorithmic techniques itself. We are not comparing performance of a various coding styles. Moreover, several today's state-of-the-art planners are provided without source code as an executable only. Hence no reasonable comparison of ideas used in the planner with other ideas is possible in such a case.

The above experimental evaluation showed that it is possible to integrate a technique known from constraint programming (namely arc-consistency) into the planning algorithm (namely into the GraphPlan algorithm) with a significant performance profit. Although this idea itself does not lead to a state-of-the-art planner it represents an interesting improvement that can be competitive in combination with other ideas and precise implementation.

If we compare our approach with other existing techniques we may see that our idea was quite original. Many techniques for solving planning problems try to directly translate the problem into another formalism. After this translation they solve the problem in a new formalism. Many of these approaches use Boolean formula (SAT) or constraint satisfaction as the target formalism. SAT based planners are described in (Kautz *et al.*, 1996; Kautz and Selman, 1999); another constraint programming methods are described in (Baioletti *et al.*, 1998; Kambhampati, 2000; Kambhampati *et al.*, 1997; Kautz and Selman, 1999; Lopez and Bacchus, 2003). The drawback of these methods is that the information induced by the original formulation is often lost during translation into the target formalism. Some planners are trying to overcome this drawback by hand tailored encoding of a planning problem into the target formalism (van Beek and Chen, 1999). We do not follow this approach.

We use constraint programming techniques to solve a small sub-problem which arises during the GraphPlan style solving process. This is in contrast to other approaches which use constraint programming formalism on the planning problem as a whole. The way how we model our problem can be viewed as a synthesis of the encoding style of the planning graph as a CSP known from GP-CSP planner (Kambhampati, 2000) and the Boolean formula satisfaction approach known from SATPlan planner (Kautz and Selman, 1999) applied in smaller scale (not for the whole problem).

We also examined what would happen if we use stronger singleton arc-consistency (Dechter, 2003; Barták and Erben, 2004; Bessière and Debruyne; 2005) instead of arc-consistency in the same situation (Surynek, 2006; Surynek, 2007a). The performed experiments showed that singleton arc-consistency is too expensive. More precisely, the usage of singleton arc-consistency significantly reduces the number of backtracks (more than the above model with maintaining arc-consistency) but the number of constraints checks increased too much. The result was that unaffordable amount of time was spent by propagating singleton arc-consistency and the overall improvement of runtime was either poor or even negative in comparison with the standard GraphPlan.

Nevertheless, we do not consider our method to be flawless. The significant drawback of the proposed maintaining arc-consistency technique for solving problems of supports is that it is only a local consistency technique. We exploit only little from the structural information hidden in the problem formulation. If we consider what a significant improvement was achieved using a local consistency technique what would be the improvement if we would have some kind of a global consistency technique? This observation led us to develop such a global technique that is described in the next section.

3.3 Advanced Constraint Model: Global Constraints

Our new global consistency is based on discovering the hidden structural information in the constraint model. We are viewing the given problem as a graph in which we search for structures. We found that complete sub-graphs represent the valuable structures with respect to the task of solving the problem of finding supports. We call the global constraint and the associated consistency based on structural decomposition of the problem *projection global constraint* and *projection consistency* respectively. The name of the concept was chosen according to consistency checking with respect to a sub-problem - we *project* the consistency into the smaller space.

3.3.1 Projection Consistency

We examined the effect of maintaining arc-consistency for solving the problem of finding supporting actions for a sub-goal in the previous section. We obtained substantive speedups using this type of reasoning compared to pure backtracking based method. However, arc-consistency provides only some type of a local reasoning over the problem. By contrast, the projection constraint introduces some type of global reasoning over the problem of finding supports. It was motivated by observation of layers of the planning graph when they are visualized as graphs (actions are vertices and mutexes are edges) - let us call such graphs *mutex graphs*. These mutex graphs embody high density of edges on majority of testing planning problems (however our method works with sparse mutex graphs as well). The high density of edges is caused by various factors. We consider that the most important factor is that certain sets of actions are intrinsically pair-wise mutually excluded (for example imagine a robot at coordinates $[3,2]$, the robot can move to coordinates in its neighborhood, so the actions are: *moveTo*$([2,2])$, *moveTo*$([2,3])$, *moveTo*$([3,3])$, …, all these actions are pair-wise mutually excluded). Such set of actions induces a complete sub-graph - a clique - within the mutex graph.

The knowledge of clique decomposition of the mutex graph allows us to do quite strong reasoning since at most one action from a clique can be selected. This is just the first

part of the idea how projection constraint works. The second part of the idea of projection constraint is to take a subset of atoms of a given sub-goal and to calculate how a certain clique of actions contributes to satisfaction of the subset of atoms. This reasoning can be used to discover that some actions within a certain clique do not contribute enough to the sub-goal and therefore can be ruled out. Actions that are ruled out are no more considered along the search and hence the search speeds up since a smaller number of alternatives must be considered.

3.3.2 Preprocessing Step: Clique Decomposition

Projection constraint assumes that a clique decomposition of a mutex graph of a given action layer of the planning graph is known. Thus we need to perform a preprocessing step in which a clique decomposition (clique cover) of the mutex graph is constructed. Let $G = (A, \mu A)$ be a *mutex graph* (vertices represent actions, edges represent action mutexes) obtained from an action layer in the planning graph (or we can interpret the action layer as a mutex graph). The task is to find a partition of the set of vertices $A = C_1 \cup C_2 \cup \ldots \cup C_n$ such that $C_i \cap C_j = \emptyset$ for every $i, j \in \{1,2,\ldots,n\} \wedge i \neq j$ and C_i is a clique with respect to μA for $i = \{1,2,\ldots,n\}$. Generally, cliques of the partitioning do not cover all the mutexes. For $mA = \mu A - (C_1^2 \cup C_2^2 \cup \ldots \cup C_n^2)$, $mA \neq \emptyset$ holds in general (where $C^2 = \{\{a,b\} \mid a,b \in C \wedge a \neq b\}$). The requirement is to minimize a pair $[n,|mA|]$ in lexicographic ordering. Unfortunately, the problem of the clique cover of the defined property is *NP*-complete on a graph without any restriction (Golumbic, 1980).

As an exponential amount of time spent in preprocessing step is unacceptable it is necessary to abandon the requirement on optimality of clique cover. Moreover, we don't know whether the mentioned optimality requirement is the right one or the best with respect to the solved problem. It is sufficient to find some clique cover to introduce projection constraint. Our experiments showed that a simple greedy algorithm provides satisfactory results. Its complexity is polynomial in the size of the input graph which is acceptable for preprocessing step.

The example of a mutex graph of the action layer of the planning graph and its clique decomposition by the greedy algorithm is shown in figure 3.8. It is the real-life mutex graph which was extracted from the action layer from the planning graph for a Dock Worker Robots problem.

The simple greedy algorithm is listed below as algorithm 3.5. The algorithm builds cliques greedily. That is, a vertex of the highest degree is always preferred. In a given graph the algorithm greedily constructs the largest clique. Then this clique is removed from the graph and is included into the clique cover. The algorithm then continues by finding the next largest clique in the remaining graph. This process is repeated until the graph is empty.

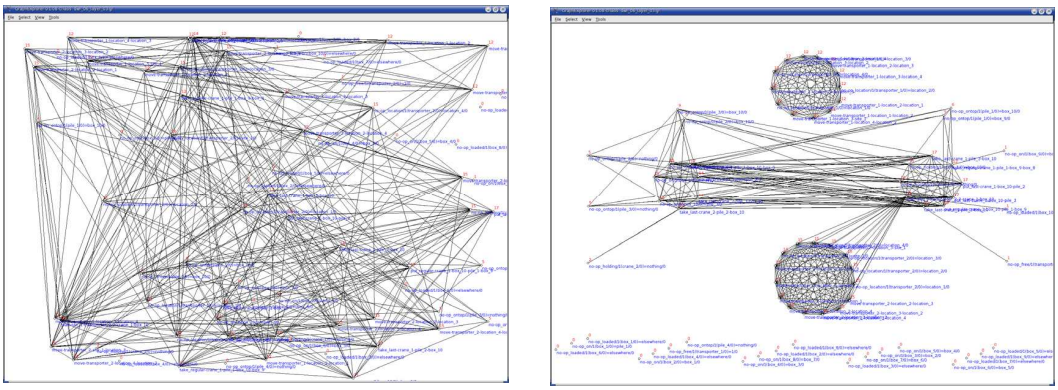


Figure 3.8. *ILLUSTRATION OF MUTEX GRAPH AND CLIQUE DECOMPOSITION.* The left part of the figure is an illustration of a mutex graph obtained from the action layer of the planning graph for a slightly more complex Dock Worker Robots problem. Vertices represent actions and edges represent action mutexes. The vertices representing actions are placed randomly in the window. The right part of the figure shows an illustration of clique decomposition of the graph on the left. The individual cliques of actions are depicted by grouping of vertices into clusters.

Algorithm 3.5. *GREEDY CLIQUE COVER ALGORITHM.* Greedy algorithm for finding clique cover of a mutex graph.

---

function *findCliqueCover*$(A, \mu A)$: **pair**

1: $n \leftarrow 1$
2: $mA \leftarrow \emptyset$
3: **while** $A \neq \emptyset$ **do**
4: $\quad C_n \leftarrow \emptyset$
5: $\quad A_n \leftarrow A$
6: $\quad \mu A_n \leftarrow \mu A$
7: $\quad$ **while** $A_n \neq \emptyset$ **do**
8: $\quad\quad a \in A_n \mid (\forall b \in A_n) \deg_{(A_n, \mu A_n)}(a) \geq \deg_{(A_n, \mu A_n)}(b)$
9: $\quad\quad C_n \leftarrow C_n \cup \{a\}$
10: $\quad\quad A_n \leftarrow \{b \in A_n \mid b \notin C_n \wedge \{a,b\} \in \mu A_n\}$
11: $\quad\quad \mu A_n \leftarrow \{\{a,b\} \mid \{a,b\} \in \mu A_n \wedge \{a,b\} \cap C_n = \emptyset\}$
12: $\quad mA \leftarrow mA \cup \{\{a,b\} \mid \{a,b\} \in \mu A \wedge |\{a,b\} \cap C_n| = 1\}$
13: $\quad A \leftarrow A - A_n$
14: $\quad \mu A \leftarrow \mu A - (C_n^2 \cup mA)$
15: $\quad n \leftarrow n + 1$
16: **return** $(\{C_1, C_2, \ldots, C_n\}, mA)$

The largest (greedy) clique itself is found by selecting the vertex of the highest degree in the current graph. The second vertex selected in the clique is the vertex of the highest

degree from the vertices neighboring to the first selected vertex. As the third vertex in the clique a vertex of the highest degree from the vertices that are neighbors of both the first selected vertex and the second selected vertex is added. This process is repeated until there are some vertices that are neighbors of all the already selected vertices.

The algorithm consists of a single function *findCliqueCover* which gets the mutex graph of some action layer of the planning graph as its parameters.

The function *findCliqueCover* gets the set of actions A (which represents vertices) and the set of action mutexes μA (which represents edges). The function returns a pair consisting of the clique cover and the set of edges (mutexes) that are outside the clique cover. The function consists of two loops - the main loop (lines 3-15) represents repeated finding of the (greedy) largest clique in the currently remaining graph. This loop is executed until the given graph is non-empty. The second loop (lines 7-11) represents repeated selection of the vertex of the highest degree. This loop is executed until the neighborhood of the selected vertices is non-empty.

Proposition 3.6 *(TIME COMPLEXITY OF THE GREEDY CLIQUE COVER ALGORITHM). The worst case time complexity of the greedy algorithm for finding clique cover (algorithm 3.5) for a graph* $G = (A, \mu A)$ *is* $O(|A|^2 + |A||\mu A|)$. ■

Proof. The internal loop of the algorithm (lines 8-11) is executed at most $O(|A|)$ times. Each iteration of the loop consumes time for finding a vertex with the highest degree and time for constructing a graph for the next iteration. Selecting a vertex of highest degree takes $O(|A|)$ steps supposed degrees of vertices are known. Construction of a graph for the next iteration (line 10-11) takes $O(|A| + |\mu A|)$ steps. In total we have the worst case time complexity of $|A|(O(|A|) + O(|A| + |\mu A|))$ which is $O(|A|^2 + |A||\mu A|)$. ■

3.3.3 Counting Derived from Clique Decomposition

For the following description we assume that a clique cover $A = C_1 \cup C_2 \cup \ldots \cup C_n$ of the set of actions A with respect to the set of mutexes μA is known. Next consider a sub-goal g we want to satisfy. Projection consistency is defined over the above decomposition for a goal $p \subseteq g$. The goal p is called a *projection goal* in this context. The projection goal represents some kind of a parameter for the new consistency. The new consistency has different results for different projection goals. The fact that at most one action from a clique can be selected allows us to introduce the following definition.

Definition 3.2 (CLIQUE CONTRIBUTION). A *contribution* of a clique $C \in \{C_1, C_2, \ldots, C_n\}$ to the projection goal p is defined as $\max(|e^+(a) \cap p| \mid a \in C)$. The contribution of the clique C to the projection goal p is denoted as $c(C, p)$. □

The concept of clique contribution is helpful when we are trying to decide whether it is possible to satisfy the projection goal using the actions from the clique cover. If for instance $\sum_{i=1}^{n} c(C_i, p) < |p|$ holds then the projection goal p cannot be satisfied. Nevertheless, the projection constraint can handle a more general case as it is described in the following definitions.

Definition 3.3 (PROJECTION CONSISTENCY: SUPPORTED ACTION). An action $a \in C_i$ for $i \in \{1,2,\ldots,n\}$ is *supported* with respect to *projection consistency* with the projection goal p if $\sum_{j=1, j\neq i}^{n} c(C_j, p) \geq |p - e^+(a)|$ holds. □

Definition 3.4 (PROJECTION CONSISTENCY: CONSISTENT PROBLEM). The preprocessed instance of the problem of finding supports consisting of actions $A = C_1 \cup C_2 \cup \ldots \cup C_n$, mutexes μA, and the goal g is *projection consistent* with respect to a projection goal $p \subseteq g$, $p \neq \emptyset$ if every action $a \in C_i$ for $i = 1,2,\ldots,n$ is supported. □

If the cliques of the clique cover are regarded as CSP variables and actions from the cliques are regarded as values for these variables then we can introduce a *projection constraint*. The projection constraint's scope contains all the clique variables. Hence the projection constraint/consistency can be regarded as a global constraint/consistency.

To enforce projection consistency with respect to some projection goal p over the problem of finding supporting actions it is necessary to rule out unsupported actions. Notice that projection consistency is not a sufficient condition to obtain a solution.

Proposition 3.7 *(CORRECTNESS OF PROJECTION CONSISTENCY). Projection consistency is correct. That is, the set of solutions of the problem of finding supporting actions S for a goal g is the same as the set of solutions of the problem of finding supporting actions S' which we obtain from S by enforcing projection consistency with respect to a projection goal $p \subseteq g$.* ■

Proof. The proposition can proved by observing that an unsupported action cannot participate in any solution. Let $a \in C_i$ be an unsupported action for some $i \in \{1,2,\ldots,n\}$. That is $\sum_{j=1, j\neq i}^{n} c(C_j, p) < |p - e^+(a)|$ holds. Observe that after the selection of the action a there is no chance to satisfy the goal p. Hence it is not possible to satisfy g since it is the superset of p. ■

A useful property of the projection consistency with a single projection goal p is that the removal of an unsupported action does not affect any of the remaining supported actions. That is if an action is supported, it remains supported after removal of any other unsupported action. We call this property a *monotonicity*. The usefulness consists in the fact

that it is enough to check each action of the problem only once to enforce the projection consistency.

Proposition 3.8 *(MONOTONICITY OF PROJECTION CONSISTENCY). Projection consistency with a projection goal p is monotone. That is, if an arbitrary unsupported a action is removed from a clique C_i for $i \in \{1,2,\ldots,n\}$ the set of supported actions within the problem remains unchanged.* ■

Proof. Let $b \in C_j$ be an unsupported action after removal of an (unsupported) action a from C_i. First, let us investigate the case when $i = j$. Action b is unsupported after removal of a if $\sum_{k=1,k\neq j}^{n} c(C_k, p) < \left|p - e^+(b)\right|$. Observe that removal of a has no effect on the truth of the expression $\sum_{k=1,k\neq j}^{n} c(C_k, p) < \left|p - e^+(b)\right|$. Hence, the action b was unsupported even before the action a was removed (removal of action a did not cause that b is unsupported).

For the case $i \neq j$ the situation is similar. Action b is unsupported after removal of a if $\sum_{k=1,k\neq i,j}^{n} c(C_k, p) + c(C_i - \{a\}, p) < \left|p - e^+(b)\right|$ *(i)*. If $c(C_i, p) = c(C_i - \{a\}, p)$ then the removal of the action a has no effect on the truth value of the expression *(i)*. If $c(C_i, p) > c(C_i - \{a\}, p)$, then $\left|p \cap e^+(a)\right| = c(C_i, p)$. From the assumption that a is unsupported we have $\sum_{k=1,k\neq i}^{n} c(C_k, p) < \left|p - e^+(a)\right|$. Hence $\sum_{k=1}^{n} c(C_k, p) < \left|p\right|$ (this indicates that the problem is unsolvable). Hence $\sum_{k=1}^{n} c(C_k, p) - c(C_j, p) < \left|p\right| - \left|e^+(b) \cap p\right| \leq \left|p - e^+(b)\right|$ and also $\sum_{k=1,k\neq j}^{n} c(C_k, p) < \left|p - e^+(b)\right|$ holds. Again we have that b was unsupported even before removal of a. ■

To discuss complexity issues of our approach we have to formally define propagation algorithm for projection consistency. The propagation algorithm for projection consistency is shown below as algorithm 3.6.

The algorithm consists of two functions - the main function *enforceProjectionConsistency* and the auxiliary function *calculateCliqueContribution*. The input of the algorithm is a projection goal p and the clique decomposition $\{C_1, C_2, \ldots, C_n\}$.

The algorithm directly follows definitions of projection consistency. First, the overall contribution γ of all the cliques of the clique cover is computed (lines 1-5). Then each action is checked whether it is supported with respect to projection consistency and the given projection goal p (lines 6-10). If the action is unsupported then it is removed from the clique decomposition (line 9).

Proposition 3.9 *(COMPLEXITY OF PROJECTION CONSISTENCY). Propagation algorithm for projection consistency with a projection goal p over the supports problem consisting of actions $A = C_1 \cup C_2 \cup \ldots \cup C_n$, mutexes μA and a goal g runs in $O(\left|p\right|\left|A\right|)$ steps.* ■

Proof. The auxiliary function *calculateCliqueContribution* performs $O(|p||C_i|)$ steps for a clique C_i for $i \in \{1,2,\ldots,n\}$ (the loop on lines 13-14 performs exactly $|C|$ iterations, each iteration of the loop takes $\Sigma|p|$ steps, where Σ is the action size bounding constant (that is $(\forall a \in A) \quad \Sigma \geq \max(|p(a)|,|e^+(a)|,|e^-(a)|)$, in fact it is sufficient to have $(\forall a \in A)$ $\Sigma \geq \max(|e^+(a)|)$). Hence lines 3-5 of the main function *enforceProjectionConsistency* take $O(\sum_{i=1}^{n}|p||C_i|) = O(|p||A|)$. Finally, the loops on lines 6-10 of the main function perform a conditional statement on the line 8 $|A|$ times. Each check of the condition in the conditional statement on the line 8 takes $\Sigma|p|$ steps. Hence we have a total number of steps $O(|p||A|)$. ■

Algorithm 3.6. *PROJECTION CONSISTENCY PROPAGATION ALGORITHM.* Projection consistency propagation algorithm with respect to a single projection goal.

function *enforceProjectionConsistency*$(\{C_1,C_2,\ldots,C_n\},p)$: **pair**
1: $\gamma \leftarrow 0$
2: *changed* $\leftarrow$ *False*
3: **for** $i = 1,2,\ldots,n$ **do**
4: | $c_i \leftarrow$ *calculateCliqueContribution*(C_i,p)
5: | $\gamma \leftarrow \gamma + c_i$
6: **for** $i = 1,2,\ldots,n$ **do**
7: | **for each** $a \in C_i$ **do**
8: | | **if** $\gamma - c_i < |p - e^+(a)|$ **then**
9: | | $C_i \leftarrow C_i - \{a\}$
10: | | *changed* $\leftarrow$ *True*
11: **return** $(changed,\{C_1,C_2,\ldots,C_n\})$

function *calculateCliqueContribution*(C,p) : **integer**
12: $c \leftarrow 0$
13: **for each** $a \in C$ **do**
14: | $c \leftarrow \max(c,|e^+(a) \cap p|)$
15: **return** c

We were not concerned about the question of how to select projection goals for a problem with a goal g until now. The only condition on a projection goal p is that $p \subseteq g$ must hold.

The projection consistency filters out different sets of inconsistent actions for different projection goals. So it is suitable to enforce projection consistency with respect to several projection goals. For maximum pruning power, we would have to enforce projection consistency for every subset of the goal g. However, this is unrealistic since there are exponentially many subsets of the goal g. Hence we can select only a limited number of projec-

tion goals. At the same time the selection must be done carefully in order to achieve strongest possible filtering effect. An analysis of projection goal selection is given below.

Let us note that enforcing consistency with respect to the projection goals is similar to *edge-finding* rules from scheduling with unary resources (Baptiste *et al.*, 2001). Edge-finding rules are defined with respect to a subset of *activities* (an action with duration). The subset of activities is used as a parameter for the rules and hence its role is similar to projection goal (it was proved that for edge-finding rules a polynomial number of subsets is sufficient).

The following ideas are focused on comparison of projection consistency with arc-consistency of the problem of finding supporting actions as it was introduced in the previous section.

Definition 3.9 (SIMPLIFIED ARC-CONSISTENCY FOR THE PROBLEM OF FINDING SUPPORTING ACTIONS). Let us have a problem of finding supporting actions S for a goal g. For each atom $t \in g$ we introduce a so called *support variable* which contains all the actions that supports the atom t in its domain (an action a *supports* an atom t if $t \in e^+(a) \wedge t \notin e^-(a)$, a set $s_t = \{a \mid a \in A \wedge \text{supports atom } t\}$ is called a *set of support*s for an atom t). Between every two support variables there is a *mutex constraint*. The mutex constraint is satisfied by an assignment of actions to its variables if the actions of the assignment are non-mutex. The supports problem is *arc-consistent* if every action mutex constraint is arc-consistent. □

Depending on the quality of the clique decomposition of the mutex graph of the supports problem there may be a situation in which a projection goal can be selected to simulate arc-consistency by projection consistency. Moreover, there may be situations when projection consistency is stronger than arc-consistency. Both cases are formally summarized in the following observations. Experiments showed that such cases are not rare, especially when projection goals are selected in order to prefer such cases.

Observation 3.1 (ARC-CONSISTENCY BY PROJECTION CONSISTENCY). *For a given problem of finding supporting actions S for a goal g there exist clique decompositions and projection goals $p \subseteq g$ such that if the problem S is projection consistent with respect to these clique decompositions and projection goals then it is arc-consistent.* ■

Proof. It is sufficient to investigate a case for a single constraint between two support variables. An action a in the domain of a support variable v should be removed in order to establish arc-consistency if it does not have a support with respect to the given constraint. That is all the actions in the domain of the support variable u which neighbors with v through the given constraint are mutex with a. Hence $a \notin D[u]$ must hold to enforce arc-consistency. Let us suppose that $\{a\} \cup D[u]$ is a part of a single action clique of the

clique decomposition. Further let us suppose that action a does not support the atom corresponding to the variable u. Then the projection consistency with respect to a projection goal p which contains exactly the atom corresponding to the variable u ($p = \{u\}$) removes action a from the clique. ■

Although the situation for the projection consistency from the proof is rather artificial, our empirical experimentation gives us evidence that it is not a rare case. Moreover, there are a lot of other similar situations when projection consistency gives the same results as arc-consistency. However, these situations are difficult to be theoretically classified (this is not our goal).

Let us note that enforcing arc-consistency by the standard *AC-3* algorithm takes $O(|g|^2|A|^3)$ steps for the problem of finding supporting actions consisting of actions from the set A (the model from the definition 3.9 is used - we have $|g|$ variables with domains of the size at most $|A|$). In contrast, the projection consistency requires only $O(|p||A|)$ steps for a single projection goal p. As we will see later even if we enforce projection consistency with respect to a certain set of projection goals, it still has lower complexity than *AC-3*.

The example of enforcing projection consistency is shown in figure 3.9. The consistency is enforced with respect to multiple projection goals in the figure.

Observation 3.2 (STRENGTH OF PROJECTION CONSISTENCY). *For a given problem of finding supporting actions S for a goal g there exists a clique decomposition and a projection goal $p \subseteq g$ such that the problem S is arc-consistent but it is not projection consistent with the projection goal p.* ■

Proof. We prove the observation by constructing an instance of the problem of finding supports. Let us have a goal $g = \{t_1, t_2, t_3\}$ where t_i for $i = 1,2,3$ are atoms and actions $a_1^1 = (\{\}, \{t_1\}, \{t_2, t_3\})$, $a_2^1 = (\{\}, \{t_2\}, \{t_1, t_3\})$, $a_3^1 = (\{\}, \{t_3\}, \{t_1, t_2\})$, $a_1^2 = (\{\}, \{t_1\}, \{t_2, t_3\})$, $a_2^2 = (\{\}, \{t_2\}, \{t_1, t_3\})$, and $a_3^2 = (\{\}, \{t_3\}, \{t_1, t_2\})$. The problem of finding supports consisting of actions $\{a_1^1, a_2^1, a_3^1, a_1^2, a_2^2, a_3^2\}$ and the goal g cannot be solved. Actions a_1^1, a_2^1 and a_3^1 are pair-wise mutex as well as actions a_1^2, a_2^2 and a_3^2. The domain of a support variable for the atom t_1 is $\{a_1^1, a_1^2\}$, for the atom t_2 it is $\{a_2^1, a_2^2\}$ and for the atom t_3 it is $\{a_3^1, a_3^2\}$. The arc-consistency on the model from the definition 3.9 does not remove any action from the domains of support variables. On the other hand, projection consistency is more successful. Suppose that the preprocessing step finds cliques $\{a_1^1, a_2^1, a_3^1\}$ and $\{a_1^2, a_2^2, a_3^2\}$. The contributions of both cliques is 1. Hence none of the actions is supported with respect to projection consistency. So the projection consistency removes all the actions and detects insolvability of the problem. ■

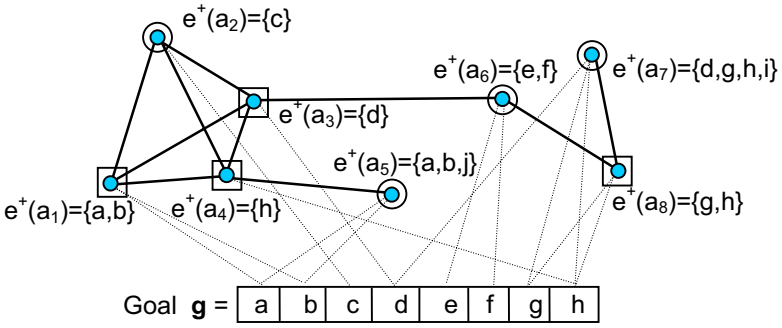


Figure 3.9. *ILLUSTRATION OF PROJECTION CONSISTENCY.* An instance of the problem of finding supporting actions consisting of eight actions. Doted lines connecting an action and an atom depict the support relation; solid lines represent mutexes. Circled vertices depict the solution of the problem (that is, the set of actions which together satisfy the goal g by their positive effects). Projection consistency with respect to multiple projection goals is enforced in this problem. Cliques detected by the greedy algorithm are: $C_1=\{a_1,a_2,a_3,a_4\}$, $C_2=\{a_6,a_8\}$, $C_3=\{a_5\}$, and $C_4=\{a_7\}$. Unsupported actions for the projection goals $p_1=\{c,e,f\}$, $p_2=\{a,b,d,g\}$, and $p_3=\{h\}$ are depicted by squared vertices. For example vertex for the action a_3 is unsupported for the projection goal $p_1=\{c,e,f\}$ since action a_3 contributes by 0, C_2 contributes by 2, C_3 contributes by 0, and C_4 contributes by 0 which is together less than the size of p_1.

In our experimental evaluation the projection consistency is enforced for projection goals $p \subseteq g$ that contains all the atoms for which the number of supporting actions (definition 3.13) is the same. More formally, let $p_i = \{t \mid t \in g \wedge |s_t| = i\}$, then projection consistency is enforced for every $i = 1,2,\ldots$ for which $p_i \neq \emptyset$.

The described selection of projection goals is partially motivated by observations 3.1 and 3.2 and partially by preliminary experiments with various variants of projection goals. Nevertheless, we do not know whether it is the best set of projection goals with respect to the ratio of pruning power and overall size.

It takes $O(\sum_{i=1,2,\ldots \,\&\, p_i \neq \emptyset} |p_i||A|) = O(|g||A|)$ steps to enforce projection consistency with respect to all projection goals as defined above. If the projection consistency is enforced with respect to one projection goal it may happen that it becomes inconsistent with respect to another projection goal. Therefore the consistency should be enforced repeatedly in the *AC-1* style (Dechter, 2003) until cliques of actions are no longer changing. This takes $O(|g||A|^2)$ (consistency enforcing is performed at most $|A|$ times since in each iteration at least one action is ruled out). It is still better than $O(|g|^2|A|^3)$ steps of *AC-3* on the model from the definition 3.9. However, the empirical tests showed that such repetition does not provide any significant extra filtering effect. Hence we use the only iteration of projection consistency with respect to projection goals p_i for $i = \{1,2,\ldots\}$ where $p_i \neq \emptyset$.

The application of the projection consistency in planning using planning graphs is very similar to the situation where we were using arc-consistency. We only need to suppose that

clique covers of action layers were constructed together with the planning graph expansion. At the point when a plan is extracted from the planning graph and more precisely at the point when the problem of finding supporting actions is resolved a constraint model is constructed.

Consider that we are solving the problem of finding supporting actions at action layer A with mutexes μA for sub-goal g. Next suppose that a clique cover of the mutex graph formed by the action layer was constructed. Let it consists of the cliques $C_1, C_2, \ldots, C_k$ and let mA be the set of mutexes outside the clique decomposition (the clique decomposition is constructed only once). We construct the constraint model consisting of variables $support(p)$ for each $p \in g$ which contains actions from A that have p as their positive effect in its domain. Next we have variables $clique(i)$ for every $i = 1,2,\ldots,k$ whose domains consist of the actions of the corresponding clique of the decomposition. Constraints in our model are represented by the set of mutexes outside the clique decomposition mA and by a single global projection constraint.

A part of the symbolic code for solving the above constraint model is listed below as algorithm 3.7. The complete symbolic code can be obtained from algorithm 3.1 by replacing the model and the call at line 15 by $propagateAdvancedModel(M', \zeta \cup \{a_{SUPPORT}\})$.

The algorithm 3.7 is again very similar to algorithms 3.2, 3.3, and 3.4. The difference here is that instead of arc-consistency the projection consistency is enforced. The meaning of the input and the output of the algorithm is the same as in the cases of algorithms 3.2, 3.3, and 3.4. The algorithm consists of a loop (lines 2-27) which is repeated until a queue of actions A_{SELECT} that are selected to satisfy the goal is non-empty. In each iteration of the loop, actions listed in the queue are included into the solution and the constraint model is modified according to the selected actions (lines 3-10). Then projection consistency is enforced (lines 11-19). The projection consistency is enforced with respect to the projection goals that contain atoms that have the same number of supporting actions (line 12). The projection consistency is enforced over the variables representing cliques of the decomposition ($clique(1), clique(2), \ldots, clique(k)$) (lines 13-14). It is then propagated to the support variables (lines 15-19). This propagation is simply done by removing each action from the domains of support variables that is missing in the domains of variables representing the clique decomposition. The constraints representing the mutexes outside the clique decomposition are utilized by enforcing arc-consistency over the set of support variables (line 20-21). After enforcing consistencies some of the support variables may have empty current domains which indicates a failure (lines 22-24) - notice that a clique variable with an empty current domain does not indicate a failure. If the algorithm passes the failure test, some of actions may become again selected (if they are the only support for a certain atom) (lines 25-26). If there exist such actions the algorithm continues with the next iteration of the main loop.

Algorithm 3.7. *CONSTRAINT PROPAGATION - PROJECTION CONSISTENCY*. This code represents the external function *propagateAdvancedModel* for the algorithm 3.1.

function *propagateAdvanceModel-Projection*(M,ζ): **pair**
1: $A_{SELECT} \leftarrow \emptyset$
2: **do**
3: **for each** $a_{SELECT} \in A_{SELECT}$ **do**
4: $h \leftarrow \{p \mid p \in e^{+}(a_{SELECT})\}$
5: $X_{DELETE} = \{support(p) \mid p \in h \wedge support(p) \in M.X\}$
6: $M.X \leftarrow M.X - X_{DELETE}$
7: $C_{DELETE} \leftarrow \{c \mid c \in M.C \wedge X_c \cap X_{DELETE} \neq \emptyset\}$
8: $M.C \leftarrow M.C - C_{DELETE}$
9: $A_{SELECT} \leftarrow A_{SELECT} - \{a_{SELECT}\}$
10: $\zeta \leftarrow \zeta \cup \{a_{SELECT}\}$
11: $g \leftarrow \{p \mid support(p) \in M.X\}$
12: **for each** $i = 1,2,\ldots$ **such that** $p_i \leftarrow \{t \mid t \in g \wedge |s_t| = i\} \neq \emptyset$ **do**
13: $(changed, \{M.D[clique(1)],\ldots,M.D[clique(k)]\}) \leftarrow$
14: $enforceProjectionConsistency(\{M.D[clique(1)],\ldots,M.D[clique(k)]\}, p_i)$
15: **if** *changed* **then**
16: $A_{REMAINING} \leftarrow M.D[clique(1)] \cup \ldots \cup M.D[clique(k)]$
17: **for each** $support(p) \in M.X$ **such that**
18: $(\exists a)(a \in support(p) \wedge a \notin A_{REMAINING})$ **do**
19: $M.D[support(p)] \leftarrow M.D[support(p)] - \{a\}$
20: $X_{SUPPORT} \leftarrow \{support(p) \mid support(p) \in M.X\}$
21: $M|_{X_{SUPPORT}} \leftarrow enforceArcConsistency\text{-}AC\text{-}3(M|_{X_{SUPPORT}})$
22: $X_{EMPTY} \leftarrow \{support(p) \in M.X|_{X_{SUPPORT}} \mid M.D[support(p)] = \emptyset\}$
23: **if** $X_{EMPTY} \neq \emptyset$ **then**
24: **return** $(M, \{failure\})$
25: **for each** $support(p) \in M.X$ **such that** $M.D[support(p)] = \{a\}$ **do**
26: $A_{SELECT} \leftarrow A_{SELECT} \cup \{active(a)\}$
27: **while** $A_{SELECT} \neq \emptyset$
28: **return** (M,ζ)

3.3.4 Experimental Evaluation

We implemented the above constraint model and projection consistency enforcing algorithm in C++ and we have integrated it into our implementation of the GraphPlan algorithm

in order to improve the solving process of the problems of finding supporting actions for a goal. For the experimental evaluation itself we used the same set of problems as in the case of evaluation of constraint models for maintaining arc-consistency. That is, we used several instances of planning problems from Dock Worker Robots environment, Towers of Hanoi environment, and from the Refueling Planes environment. Again the same statistical characteristics were collected. The tests were performed on the same hardware (two AMD Opterons 242 - 1600 MHz with 1GB of memory) as previous experimental evaluation. Our implementation was again compiled using the gcc compiler version 3.4.3 with maximum optimization for the target machine (-O3 -mtune=opteron). The tests were again run under Mandriva Linux 10.2. These settings of the experiments allow us to directly compare performance of the standard GraphPlan algorithm and the variants of maintaining arc-consistency with the just proposed application of maintaining projection consistency in terms of time.

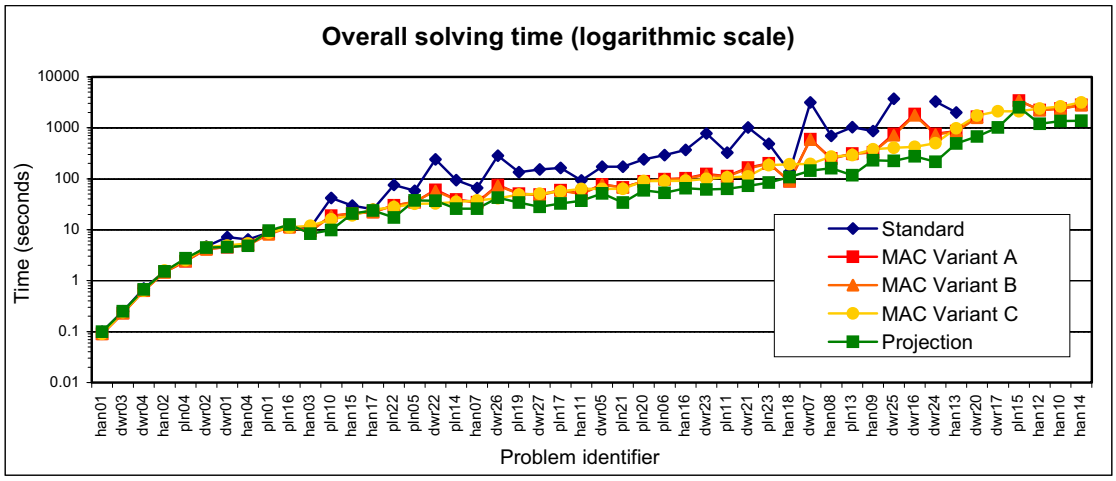


Figure 3.10. *COMPARISON OF OVERALL SOLVING TIMES (LOGARITHMIC SCALE) - (STD, VARA, VARB, VARC, PRJ).* Comparison of the overall solving time of the standard GraphPlan algorithm and enhanced versions which use maintaining arc-consistency and maintaining projection consistency for solving the problem of finding supports (standard version and variants *A*,*B*, and *C* and projection consistency propagation schemes are compared). Problems on the horizontal axis are listed in the ascending order according to the time consumed by the *variant C*. Time limit of 1 hour for each problem is used.

Since the projection consistency was further improved we postpone the presentation of complete results into the next section. We show the overall solving time comparison of the algorithm that uses projection consistency with all the previously discussed algorithms. The results in figure 3.10 shows the comparison of the standard version of the GraphPlan algorithm and the enhanced versions which use maintaining arc-consistency of variants *A*, *B*, and *C* and maintaining projection consistency. The ordering of problems along the horizon-

tal axis is according to the ascending time consumed by the *variant C*. The results show that the version with maintaining projection consistency is the best on almost all the tested problems. The projection consistency loses with the *variant C* more significantly only on one problem (problem pln15). Notice that projection consistency requires an extra time for building the clique cover and still it is faster.

3.3.5 Conclusion and Discussion

We proposed a novel consistency technique which we called projection consistency. The technique is designed to prune the search space during extraction of plans by the GraphPlan algorithm. We theoretically showed that the projection consistency has faster propagation algorithm than the arc-consistency propagation algorithm AC-3 when applied on the same problem. Empirical tests showed significant improvements compared to the standard GraphPlan and also compared to the version using arc-consistency.

Experimental comparison of projection consistency with local arc-consistency technique confirmed the hypothesis that global consistency that respects the structural information encoded in the problems provides better propagation. However, there are still some questions regarding the proposed technique.

The first interesting issue is how to make projection consistency stronger. This may be done by other types of projection goals. But it is also possible to do it by the slight modification of the definition of the supported action. Instead of the expression $\sum_{j=1,j\neq i}^{n} c(C_j, p) \geq \left|p - e^+(a)\right|$ in the definition 3.12 one can use $\sum_{j=1,j\neq i}^{n} c(C_j, p - e^+(a)) \geq \left|p - e^+(a)\right|$. Unfortunately this change causes that monotonicity (proposition 3.8) - the main argument for low complexity of propagation algorithm - no longer holds.

The similarity between Boolean formula satisfaction problem and the problem of finding supporting actions for a goal as it is shown in proposition 3.1 leads us to the question whether it is possible to exploit projection consistency for solving SAT problems. We deal with this question in the chapter 4. The expectable question is also how to extend the presented ideas for planning graphs with time and resources (Long and Fox, 2003; Smith and Weld, 1999). Since the planning graphs for complex problems are really large the related question is also how to make planning graphs unground and how to get rid of high numbers of no-operation actions.

3.4 Tractable Class of Problem of Finding Supports

We discuss a special class of the problem of finding supporting actions with respect to proposed projection consistency in this section. Consider that we have a clique decomposition

of the mutex graph of a certain action layer of the planning graph. Next consider that a set of atoms supported by actions in the clique is constructed for each clique. We noticed that the intersection graph (Golumbic, 1980), where vertices are these sets and edges are their non-empty intersections, has typically a simple structure resembling interval graphs.

The method described in this section is trying to utilize the above observation for solving the problem of supporting actions in connection with projection consistency. More precisely, we present a polynomial time solving procedure for solving the problem of supports when the above intuitively defined graph is acyclic. Next, we propose a heuristic that guides the solving process of the general problem of supports that is trying to simplify the problem to one belonging into the tractable class.

3.4.1 Tractability

It is possible to make projection consistency stronger by a slight reformulation of the definition of the supported action. The definition of the consistent problem remains the same. We will need the modified version of the projection consistency to be able to solve certain instances of the problem of finding supporting actions in polynomial time.

Definition 3.10 (STRONGLY SUPPORTED ACTION). An action $a \in C_i$ for $i \in \{1,2,\ldots,n\}$ is *strongly supported* with respect to *projection consistency* with the projection goal p if $\sum_{j=1, j\neq i}^{n} c(C_j, p - e^+(a)) \geq |p - e^+(a)|$ holds. □

Let us call the projection consistency that uses the definition of strongly supported actions a *strong projection consistency.* The new variant of consistency is correct.

Proposition 3.10 *(CORRECTNESS OF STRONG PROJECTION CONSISTENCY). Strong projection consistency is correct. That is, the set of solutions of the problem of finding supporting actions S for a goal g is the same as the set of solutions of the problem of finding supporting actions S' which we obtain from S by enforcing strong projection consistency with respect to a projection goal $p \subseteq g$.* ■

Proof. As in the proof of correctness of the basic variant of projection consistency we show that an unsupported action cannot participate in any solution. Let $a \in C_i$ be an unsupported action for some $i \in \{1,2,\ldots,n\}$. That is $\sum_{j=1, j\neq i}^{n} c(C_j, p - e^+(a)) < |p - e^+(a)|$ holds. Observe that after the selection of the action a it is not possible to satisfy the goal p. Hence it is not possible to satisfy g since it is the superset of p. ■

Proposition 3.11 *(STRONGER PROJECTION CONSISTENCY). If the problem of supports is strongly projection consistent with respect to a projection goal p then it is projection con-*

sistent with respect to the projection goal p. Moreover there exists a the problem of supports which is projection consistent with respect to a projection goal p and it is not strongly projection consistent with respect to the same projection goal p. ■

Proof. To prove the first part of the proposition it is sufficient to observe that $\sum_{j=1,j\neq i}^{n} c(C_j, p-e^+(a)) \geq |p-e^+(a)| \Rightarrow \sum_{j=1,j\neq i}^{n} c(C_j,p) \geq |p-e^+(a)|$ for any $a \in C_i$ for $i=\{1,2,\ldots,n\}$ and for any projection goal p. Moreover, there exists a problem of finding supports and the projection goal p where for some $a \in C_i$ and $i \in \{1,2,\ldots,n\}$ inequalities $\sum_{j=1,j\neq i}^{n} c(C_j,p) \geq |p-e^+(a)|$ and $\sum_{j=1,j\neq i}^{n} c(C_j, p-e^+(a)) < |p-e^+(a)|$ hold. Let $C_1=\{a_1\}$ and $C_2=\{a_2,a_3\}$ where $a_1=(\{\},\{\heartsuit,\spadesuit\},\{\})$ $a_2=(\{\},\{\clubsuit,\spadesuit\},\{\})$, $a_3=(\{\},\{\diamondsuit,\heartsuit\},\{\})$. If we take $p=\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}$ and action $a_2 \in C_2$ then $\sum_{j=1,j\neq 2}^{n} c(C_j,p) = c(C_1,\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}) = 2 \geq |p-e^+(a_2)| = |\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}-\{\heartsuit,\spadesuit\}| = 2$. However $\sum_{j=1,j\neq 2}^{n} c(C_j, p-e^+(a_2)) = c(C_1,\{\diamondsuit,\heartsuit\}) = 1 < |p-e^+(a_2)| = |\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}-\{\heartsuit,\spadesuit\}| = 2$. ■

The modification of the definition was simple. Unfortunately this is not true for the propagation algorithm. Our modification substantially changed the effect of removal of an unsupported action on the set of strongly supported actions with respect to a single projection goal. The set of supported actions does not change after removal of an unsupported action in case of projection consistency. This property of projection consistency is called a *monotonicity* and represents the main argument for the low complexity of the propagation algorithm. For strong projection consistency the monotonicity does not hold (the set of supported actions may change). Fortunately, this property does not matter for the (tractable) case we are about to investigate.

Proposition 3.12 *(NON-MONOTONICITY OF STRONGER PROJECTION). Strong projection consistency with a projection goal p is not monotone. That is, there exists a problem where a supported action a becomes unsupported after the removal of another unsupported action b (support relations are considered with respect to p).* ■

Proof. Let us have a problem with the goal $g=\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit,\Diamond,\nabla\}$ and a clique decomposition $C_1=\{a_1\}$, $C_2=\{a_2\}$, and $C_3=\{a_3\}$ where $a_1=(\{\},\{\clubsuit,\diamondsuit\},\{\})$ $a_2=(\{\},\{\diamondsuit,\heartsuit\},\{\})$, $a_3=(\{\},\{\Diamond,\nabla\},\{\})$. Let $p=g$. The action a_1 is unsupported with respect to p since $\sum_{j=1,j\neq 1}^{n} c(C_j, p-e^+(a_1)) = c(C_2,\{\heartsuit,\spadesuit,\Diamond,\nabla\}) + c(C_3,\{\heartsuit,\spadesuit,\Diamond,\nabla\}) = 1+2=3 < |p-e^+(a_1)| = |\{\heartsuit,\spadesuit,\Diamond,\nabla\}| = 4$. Similarly for the remaining actions. The action a_2 is also unsupported with respect to p since $\sum_{j=1,j\neq 2}^{n} c(C_j, p-e^+(a_2)) = c(C_1,\{\clubsuit,\spadesuit,\Diamond,\nabla\}) + c(C_3,\{\clubsuit,\spadesuit,\Diamond,\nabla\}) = 1+2=3 < |p-e^+(a_2)| = |\{\clubsuit,\spadesuit,\Diamond,\nabla\}| = 4$. The action a_3 is supported with respect to p since $\sum_{j=1,j\neq 3}^{n} c(C_j, p-e^+(a_3)) = c(C_1,\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}) + c(C_2,\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}) = 2+2=4 \geq |p-e^+(a_3)| = |\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}| = 4$. After removal of the unsupported action a_1 the action a_3 becomes also unsupported since $c(C_1-\{a_1\}, p-e^+(a_3)) + \sum_{j=1,j\neq 1,3}^{n} c(C_j, p-e^+(a_3)) = c(C_1-\{a_1\},\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}) + c(C_3,\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}) = 0+2=2 < |p-e^+(a_3)| = |\{\clubsuit,\diamondsuit,\heartsuit,\spadesuit\}| = 4$. ■

Definition 3.11 (MERGED POSITIVE EFFECT). For a clique $C \in \{C_1, C_2, \ldots, C_n\}$ of the action clique decomposition we define a *merged positive effect* as $\bigcup_{a \in C} e^+(a)$. It is denoted as $me^+(C)$.□

Definition 3.12 (CLIQUE INTERSECTION GRAPH). We define a *clique intersection graph* $G_I = (\{C_1, C_2, \ldots, C_n\}, E_I)$ for the action clique decomposition $A = C_1 \cup C_2 \cup \ldots \cup C_n$ as an undirected intersection graph of corresponding merged positive effects. That is $E_I = \{\{C_i, C_j\} \mid i \neq j \wedge me^+(C_i) \cap me^+(C_j) \neq \emptyset\}$. □

Proposition 3.13 *(TRACTABLE CASE: PROJECTION CONSISTENCY). Let* $A = C_1 \cup C_2 \cup \ldots \cup C_n$ *be a clique decomposition of the action layer and let* g *be a goal we want to satisfy. Next let* $G_I = (V_I, E_I)$ *be the corresponding clique intersection graph. If the graph* G_I *is acyclic then a problem of satisfying the goal* g *by selecting just one action* a_i *from the clique* C_i *for every* $i = 1, 2, \ldots, n$ *can be solved in polynomial time after enforcing strong projection consistency with respect to certain projection goals.* ■

Proof. We need to show that if the defined problem is strong projection consistent with respect to the certain projection goals then it is necessary to do only little to find the solution or conclude that there is no solution.

Let us take the projection goals $g \cap (me^+(C_i) - \bigcup_{j=1,\, j \neq i}^{n} me^+(C_j))$ for every $i = 1, 2, \ldots, n$ and $g \cap me^+(C_i) \cap me^+(C_j)$ for every $\{C_i, C_j\} \in E_I$. If $g - \bigcup_{i=1}^{n} me^+(C_i) \neq \emptyset$ holds then there is obviously no solution. This condition can be checked in $\sum O(|g||A|)$ steps, where Σ is the action size bounding constant (that is $(\forall a \in A)\ \Sigma \geq \max(|p(a)|, |e^+(a)|, |e^-(a)|)$). If $g \subseteq \bigcup_{i=1}^{n} me^+(C_i)$ holds then arbitrary selection of just one action a from the clique C_i for every $i = 1, 2, \ldots, n$ which preserves relation of strong supports over the edges E_I solves the problem. This selection can be done by starting in the root clique of G_I and continuing to the leaves according to the breadth first order. It takes $O(|g||A|)$ steps to select actions in this way.

Consider an arbitrary atom $t \in g$. There are at most two cliques for which the atom t is an element of their merged effect. This is due to the acyclicity of the corresponding clique intersection graph G_I. In the case when there is just one such clique C_i we show that an arbitrary selection of an action $a \in C_i$ satisfies t. Let $p = g \cap (me^+(C_i) - \bigcup_{j=1,\, j \neq i}^{n} me^+(C_j))$, for such p we have $t \in p$ and $\sum_{j=1, j \neq i}^{n} c(C_j, p - e^+(a)) \geq |p - e^+(a)|$ since the problem is strong projection consistent with respect to the projection goal p. We also have $\sum_{j=1, j \neq i}^{n} c(C_j, p - e^+(a)) = 0$ since the sum is empty (no other clique intersects the projection goal p by its merged effect). Hence $|p - e^+(a)| = 0$ and $t \in e^+(a)$, that is, we do not need to care about satisfaction of such atoms. Let us investigate the case when there are two cliques C_i and C_j for which $t \in me^+(C_i)$ and $t \in me^+(C_j)$. Suppose that an action a is selected from the clique C_i and an action b from the clique C_j. Consider the projection goal $p = g \cap me^+(C_i) \cap me^+(C_j)$,

both actions are strongly supported with respect to p. That is $\sum_{k=1,k\neq i}^{n} c(C_k, p-e^+(a)) \geq |p-e^+(a)|$ and $\sum_{k=1,k\neq j}^{n} c(C_k, p-e^+(b)) \geq |p-e^+(b)|$. Without loss of generality suppose that action a was selected before b. Since there are only two cliques interfering over the projection goal p, we specially have $c(C_j, p-e^+(a)) \geq |p-e^+(a)|$ after selecting a (the sum on the left reduces to the single summand). Hence it is possible to select the action b from C_j such that $|(p-e^+(a)) \cap e^+(b)| = c(C_j, p-e^+(a))$. Altogether we obtained that $p \subseteq e^+(a) \cup e^+(b)$, hence $t \in e^+(a) \cup e^+(b)$. ■

The question arises whether the strong projection consistency with respect to the projection goals mentioned in the proof of the proposition 3.13 can be enforced over the acyclic problem in polynomial time.

We use the idea that is used to enforce arc-consistency in an acyclic constraint network (Dechter, 2003). It is possible to enforce arc-consistency in such a network by enforcing directed arc-consistency in the direction from the leaves to the root of the network and then from the root to the leaves according to the breath-first search ordering of the network. Almost the same can be done for the strong projection consistency. First we enforce the consistency for the projection goals $g \cap (me^+(C_i) - \bigcup_{j=1,\, j\neq i}^{n} me^+(C_j))$ for every $i = 1,2,\ldots,n$. Then cliques of the decomposition are ordered according to the breadth-first search and the strong projection consistency is enforced over the edges of the intersection graph. It is done in the direction from the leaves to the root of the clique intersection graph first and then from the root to the leaves.

The complete algorithm for enforcing strong projection consistency with respect to the discussed projection goals is shown here as algorithm 3.8. Let us briefly describe the symbolic code of the algorithm. The strong consistency propagation algorithm for the acyclic clique intersection graph consists of three functions - *enforceStrongProjectionConsistency*, *propagateStrongProjection*, and *breadthFirstSearch*. The function *enforceStrongProjectionConsistency* is the main function and *propagateStrongProjection* and *breadthFirstSearch* are auxiliary functions.

The function *enforceStrongProjectionConsistency* gets the goal g and $\{C_1, C_2, \ldots, C_n\}$ clique decomposition of the mutex network. The strong projection consistency is first enforced over the leaves of the clique intersection graph (lines 3-6) with respect to special projection goals. Then the clique intersection graph is ordered according to the breath first search (line 7) and strong projection consistency is enforced over the edges of the clique intersection graph in the direction from the leaves to the root (lines 8-12) and in the opposite direction from root to the leaves (lines 13-17). Special projection goals corresponding to the edges of the clique intersection graphs are used. The return value of the function is a pair of the modified cliques of the decomposition and the indicator of the change.

The function *propagateStrongProjection* makes the problem strong projection consistent with respect to a single projection goal. The function gets the projection goal and the clique decomposition as its parameters and returns the modified clique decomposition to-

gether with the indicator of change. The last function *breadthFirstSearch* returns the numbering of vertices of a give graph in breath-first order. The resulting numbering is returned in the form of a permutation.

Algorithm 3.8. *CONSTRAINT PROPAGATION - STRONG PROJECTION CONSISTENCY*. Strong projection consistency propagation algorithm for acyclic clique intersection graph.

---

function $enforceStrongProjectionConsistency(g,\{C_1,C_2,\ldots,C_n\})$: **pair**
1: $changed \leftarrow False$
2: **let** $G_I=(\{C_1,C_2,\ldots,C_n\},E_I)$ be the clique intersection graph
3: **for** $i=1,2,\ldots,n$ **do**
4: $\quad p \leftarrow g \cap (me^+(C_i)-\bigcup_{j=1,\,j\neq i}^{n} me^+(C_j))$
5: $\quad (\chi,\{C_i\}) \leftarrow propagateStrongProjection(p,\{C_i\})$
6: $\quad changed \leftarrow changed \vee \chi$
7: $\pi \leftarrow breadthFirstSearch(G_I)$
8: **for** $j=n,n-1,\ldots,2$ **do**
9: $\quad$ **let** $\{C_{\pi(i)},C_{\pi(j)}\} \in E_I$ **such that** $\pi(i)<\pi(j)$
10: $\quad p \leftarrow g \cap me^+(C_{\pi(i)}) \cap me^+(C_{\pi(j)})$
11: $\quad (\chi,\{C_{\pi(i)},C_{\pi(j)}\}) \leftarrow propagateStrongProjection(p,\{C_{\pi(i)},C_{\pi(j)}\})$
12: $\quad changed \leftarrow changed \vee \chi$
13: **for** $i=1,2,\ldots,n-1$ **do**
14: $\quad$ **for each** $\{C_{\pi(i)},C_{\pi(j)}\} \in E_I$ **such that** $\pi(i)<\pi(j)$ **do**
15: $\quad\quad p \leftarrow g \cap me^+(C_{\pi(i)}) \cap me^+(C_{\pi(j)})$
16: $\quad\quad (\chi,\{C_{\pi(i)},C_{\pi(j)}\}) \leftarrow propagateStrongProjection(p,\{C_{\pi(i)},C_{\pi(j)}\})$
17: $\quad\quad changed \leftarrow changed \vee \chi$
18: **return** $(changed,\{C_1,C_2,\ldots,C_n\})$

function $propagateStrongProjection(p,\{C_1,C_2,\ldots,C_m\})$: **pair**
19: $\chi \leftarrow False$
20: **for** $i=1,2,\ldots,m$ **do**
21: $\quad$ **for each** $a \in C_i$ **do**
22: $\quad\quad$ **if** $\sum_{j=1,j\neq i}^{n} c(C_j, p-e^+(a)) < |p-e^+(a)|$ **then**
23: $\quad\quad\quad C_i \leftarrow C_i-\{a\}$
24: $\quad\quad\quad \chi \leftarrow True$
25: **return** $(\chi,\{C_1,C_2,\ldots,C_m\})$

function $breadthFirstSearch((\{v_1,v_2,\ldots,v_m\},E))$: **permutation**
26: $Q \leftarrow []$
27: $F \leftarrow \emptyset$
28: $k \leftarrow 1$

29: **for** $i = 1,2,\ldots,m$ **do**
30: | **if** $v_i \notin F$ **then**
31: | | $\pi(i) \leftarrow k$
32: | | $k \leftarrow k+1$
33: | | $Q \leftarrow concatenate(Q, v_i)$
34: | | $F \leftarrow F \cup \{v_i\}$
35: | | **while** $Q \neq []$ **do**
36: | | | $[u \mid R] \leftarrow Q$
37: | | | $Q \leftarrow R$
38: | | | **for each** $\{u, v_j\} \in E$ **do**
39: | | | | **if** $v_j \notin F$ **then**
40: | | | | | $\pi(j) \leftarrow k$
41: | | | | | $k \leftarrow k+1$
42: | | | | | $Q \leftarrow concatenate(Q, v_j)$
43: | | | | | $F \leftarrow F \cup \{v_j\}$
44: **return** π

The relatively simple structure of the clique intersection graph and the selected set of projection goals allow us to compute the strong projection consistency of the given problem (we are overcoming the fact that monotonicity does not hold). The key idea was to reduce the effect of removal of an unsupported action to the remaining actions with respect to the projection goals. In other words, if we remove an unsupported action only the actions from the cliques neighboring through an edge in clique intersection graph can become unsupported with respect to the selected projection goals.

The following proposition summarizes the claim that algorithm 3.8 enforces strong projection consistency with respect to the selected projection goals.

Proposition 3.14 *(CORRECTNESS OF STRONG PROJECTION CONSISTENCY ENFORCING ALGORITHM). Let* $A = C_1 \cup C_2 \cup \ldots \cup C_n$ *be a clique decomposition of the action layer and let* g *be a goal we want to satisfy. Next let* $G_I = (V_I, E_I)$ *be the corresponding clique intersection graph. If the graph* G_I *is acyclic then strong projection consistency with respect to the goals* $g \cap (me^+(C_i) - \bigcup_{j=1,\, j\neq i}^{n} me^+(C_j))$ *for every* $i = 1,2,\ldots,n$ *and* $g \cap me^+(C_i) \cap me^+(C_j)$ *for every* $\{C_i, C_j\} \in E_I$ *is enforced by algorithm 3.8.* ■

Proof. Enforcing strong projection consistency for projection goals $p^i = g \cap (me^+(C_i) - \bigcup_{j=1,\, j\neq i}^{n} me^+(C_j))$ for every $i = 1,2,\ldots,n$ means that $p^i \subseteq e^+(a)$ must hold for every $a \in C_i$ and for every $i = 1,2,\ldots,n$. The projection goals of this form are disjoint ($p^i \cap p^j = \emptyset$ for $i, j = 1,2,\ldots,n \wedge i \neq j$) and only one clique can contribute to such a projection goal (only C_i contributes to p^i). Hence it is sufficient to check actions in C_i for strong projection consistency (each action is checked once) only with respect to projection goal p^i for every

$i = 1,2,\ldots,n$ (for actions $b \in C_j$ where $j \neq i \;\; p^i \cap e^+(b) = \emptyset$ holds). Individual cliques and their actions can be considered in an arbitrary order since removal of an unsupported action with respect to projection goal p^i for some $i = 1,2,\ldots,n$ does not influence relation of strong supporting of any other action with respect to any projection goal p^j for $j = 1,2,\ldots,n$.

For enforcing strong projection consistency with respect to a single projection goal $p^{i,j} = g \cap me^+(C_i) \cap me^+(C_j)$ where $\{C_i, C_j\} \in E_I$ it is sufficient to check (and remove) actions in C_i and in C_j (each action is checked once). If an action $a \in C_i$ is strongly supported with respect to $p^{i,j}$, it means that there is an action $b \in C_j$ such that $p^{i,j} \subseteq e^+(a) \cup e^+(b)$. Hence a supported action $a \in C_i$ with respect to $p^{i,j}$ cannot be made unsupported with respect to $p^{i,j}$ by removing any unsupported action $b \in C_j$ with respect to $p^{i,j}$. The removal of an action $a \in C_i$ (unsupported with respect to other projection goal than $p^{i,j}$) can make unsupported with respect to $p^{i,j}$ only actions in the clique C_j (for actions $b \in C_k$ where $k \neq i \wedge k \neq j \;\; p^{i,j} \cap e^+(b) = \emptyset$ holds since otherwise G_I has a cycle).

Let us now consider the consistency with respect to all the projection goals $p^{i,j}$ where $\{C_i, C_j\} \in E_I$. It is sufficient to enforce strong projection consistency for projection goals corresponding to the edges of G_I in the reverse breadth-first search order and then in the normal breadth-first search order. Let π be a breadth-first search ordering of G_I with $C_{\pi(1)}$ as the root. In the first phase (lines 8-12) we proceed from the leaves to the root while actions unsupported with respect to the direct successor clique (with respect to π) are removed. That is, if $C_{\pi(j)}$ is a direct successor of $C_{\pi(i)}$ for $i = 1,2,\ldots,n-1$ then actions unsupported with respect to $p^{\pi(i),\pi(j)}$ are removed from $C_{\pi(i)}$. At the end of this phase for every action $a \in C_{\pi(i)}$ for $i = 1,2,\ldots,n-1$ there is an action $b \in C_{\pi(j)}$ for every successor $C_{\pi(j)}$ of $C_{\pi(i)}$ such that $p^{\pi(i),\pi(j)} \subseteq e^+(a) \cup e^+(b)$ *(i)*. Notice, that when removing an action from $C_{\pi(i)}$ to enforce the consistency with respect to $p^{\pi(i),\pi(j)}$, an action in another successor (a successor that has been already treated) of $C_{\pi(k)}$ of $C_{\pi(i)}$ may become unsupported.

Hence, we need to enforce that for every action $b \in C_{\pi(j)}$ for $j = 2,3,\ldots,n$ there is an action $a \in C_{\pi(i)}$ for a direct predecessor $C_{\pi(i)}$ of $C_{\pi(j)}$ such that $p^{\pi(i),\pi(j)} \subseteq e^+(a) \cup e^+(b)$ *(ii)*. Actions unsupported with respect to their direct predecessors are removed in the second phase (lines 13-17). We proceed from the root to the leaves in the second phase. To enforce *(ii)* we remove unsupported actions from $C_{\pi(j)}$ for $j = 2,3,\ldots,n$ with respect to $p^{\pi(i),\pi(j)}$ where $C_{\pi(i)}$ is a direct predecessor of $C_{\pi(j)}$. Notice that *(i)* remains unaffected since there is always only one predecessor with respect to the breadth-first search ordering of cliques. Hence at the end of the second phase *(i)* and *(ii)* hold. That is, the problem is strongly projection consistency with respect to all the projection goals $p^{i,j}$ where $\{C_i, C_j\} \in E_I$. ■

Let us note that the situation in the proof is similar to arc-consistency when it is enforced in the acyclic constraint network (Dechter, 2003). The following proposition summarizes the complexity of the algorithm for enforcing the strong projection consistency.

Proposition 3.15 *(COMPLEXITY OF STRONG PROJECTION CONSISTENCY). The algorithm for enforcing strong projection consistency over an action clique decomposition* $A = C_1 \cup C_2 \cup \ldots \cup C_n$ *which has an acyclic clique intersection graph has the worst case time complexity of* $O(|g||A|)$. ■

Proof. The function *propageStrongProjection* takes $O(|p^i||C_i|)$ steps for projection goal $p^i = g \cap (me^+(C_i) - \bigcup_{j=1,\, j\neq i}^{n} me^+(C_j))$ and $O(|p^{i,j}|(|C_i|+|C_j|))$ for projection goal $p^{i,j} = g \cap me^+(C_i) \cap me^+(C_j)$. In total, we need $O(\sum_{i=1}^{n}|p^i||C_i|)$ steps (which is $O(|g||A|)$) for projection goals of the first form and $O(\sum_{\{i,j\}\in E_I}|p^{i,j}|(|C_i|+|C_j|))$ steps (which is again $O(|g||A|)$ since the clique intersection graph is acyclic) for projection goals of the second form. Hence the total time consumed by the function *propageStrongProjection* is $O(|g||A|)$. The breadth first search performed over the clique intersection graph (line 7) takes $O(n)$ steps which is again $O(|g||A|)$. The overall time complexity is thus $O(|g||A|)$. ■

Definition 3.12 (MUTEX NETWORK). A *mutex network* for the action clique decomposition $A = C_1 \cup C_2 \cup \ldots \cup C_n$ and for the set of mutexes outside the decomposition mA is a graph $G_m = (\{C_1, C_2, \ldots, C_n\}, E_m)$, where $E_m = \{\{C_i, C_j\} \mid i \neq j \wedge (\exists a_i \in C_i, \exists a_j \in C_j)\{a_i, a_j\} \in mA\}$. □

Proposition 3.16 *(TRACTABLE CASE: MUTEX NETWORK). Let us have a clique decomposition of the action layer of the planning graph* $A = C_1 \cup C_2 \cup \ldots \cup C_n$ *and a set of mutexes outside the clique decomposition* mA. *Let* $G_m = (V_m, E_m)$ *be the corresponding mutex network. If the graph* G_m *is acyclic then a problem of selecting just one action* a_i *from the clique* C_i *for every* $i = 1,2,\ldots,n$ *such that no two selected actions are mutex with respect to* mA *can be solved in polynomial time.* ■

Proof. This is a well known result from constraint programming in fact. If each clique of the clique decomposition $A = C_1 \cup C_2 \cup \ldots \cup C_n$ is regarded as a CSP variable and mutexes of the set mA are regarded as constraints then the defined problem of selecting non-mutex actions is an acyclic constraint satisfaction problem. It is sufficient to enforce arc-consistency and to label the variables in breadth first order to obtain a solution. More details about this result can be found in (Dechter, 2003). The arc-consistency algorithm runs in polynomial time with respect to $|A|$ and $|mA|$. ■

Proposition 3.17 *(OVERALL TRACTABLE CASE). Let us have a clique decomposition of the action layer of the planning graph* $A = C_1 \cup C_2 \cup \ldots \cup C_n$ *and a set of mutexes outside the clique decomposition* mA. *Let* $G_I = (V_I, E_I)$ *be the corresponding clique intersection graph and let* $G_m = (V_m, E_m)$ *be the corresponding mutex network. If the graph* $G = (\{C_1, C_2, \ldots, C_n\}, E_I \cup E_m)$ *is acyclic then the corresponding problem of finding supports can be solved in polynomial time.* ■

Proof. To prove the proposition we use a combination of results from propositions 3.13, 3.14, 3.15 and 3.16. The first step consists of enforcing strong projection consistency and arc-consistency in the problem of finding supports. Since it is quite easy using the above results we describe the process briefly. If the interference of cliques is through an edge from E_I then strong projection consistency is enforced over the intersection of the corresponding merged positive effects. If the interference of cliques is through an edge from E_m then arc-consistency with respect to mA is enforced. Again this combined consistency can be enforced in polynomial time by proceeding from the leaves to the root of the graph G and conversely. The extraction of a solution from the consistent problem can be also done in polynomial time. The extraction procedure starts by selecting an action from the root clique and proceeds to the leaves of the graph G while strong projection consistency and arc-consistency relations are preserved over the edges of G. The described solution extraction can be carried out in polynomial time. ■

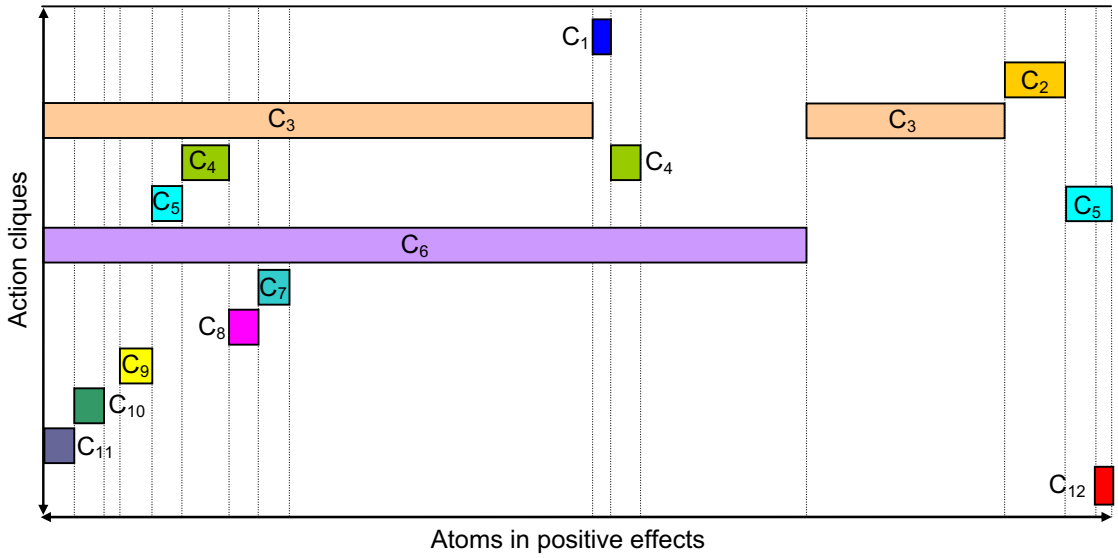


Figure 3.11. *A DIAGRAM OF MERGED POSITIVE EFFECTS OF THE CLIQUE DECOMPOSITION.* A diagram of merged positive effects of cliques of action layer clique decomposition. Each line of the diagram represents a clique. The scope of the merged positive effect of the clique is depicted as one or more horizontal bars. The width of bars is proportional to the number of actions in the individual action cliques. The diagram was constructed according to an action layer of the planning graph for an instance of the Dock Worker Robots domain.

We described the tractable class of the problem of finding the supporting actions for a goal in order to utilize the theoretical results in solving real problems. The obstacle is that not every instance of the problem of supports belongs to the described class. The figure 3.8 shows a real example of the clique decomposition of the action layer of the planning graph of an instance of the Dock Worker Robots planning domain. The corresponding diagram of merged positive effects is shown in figure 3.11 and the corresponding clique intersection

graph is shown in figure 3.12. According to these figures the problem does not belong to just defined class (the graph of the figure 3.12 is not acyclic). However, the problem is very close to our tractable class. The clique intersection graph can be made acyclic by removing a single vertex. The vertex removal corresponds to the selection of an action from the clique corresponding to the vertex, namely C_6. After having the clique intersection graph acyclic we can use the algorithm described above.

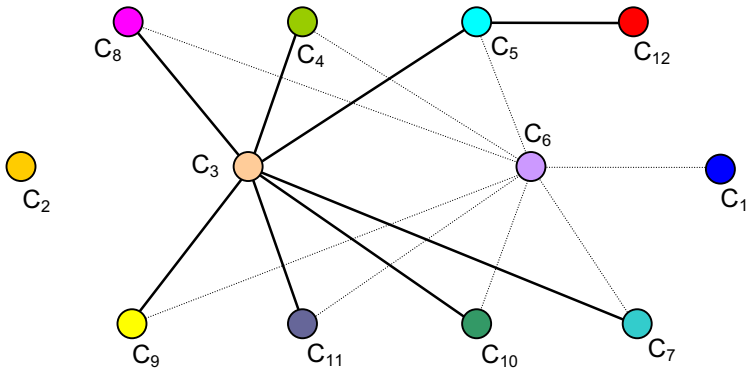


Figure 3.12. *CLIQUE INTERSECTION GRAPH.* An intersection graph of merged positive effects of cliques of the action layer clique decomposition form figure 3.11. The effect of removing of the cycle-cut-set consisting of the vertex C_6 is denoted by doted edges.

The obstacle here is that determining the smallest set of vertices (cycle-cut-set) which removal makes the graph acyclic is *NP*-complete (Dechter, 2003). So, we cannot afford to solve the problem of cycle-cut-set optimally. For our purposes we do not need an optimal cycle-cut-set but the smaller the cycle-cut-set is the larger is the complement that can be solved in backtrack-free manner by the proposed polynomial time solving algorithm.

For selecting actions we suggest to use highest degree heuristics. That is an action from the clique of the highest degree of the clique intersection graph (merged with corresponding mutex network) is selected preferably. Notice, that the clique of the highest degree in the clique intersection graph is often the largest clique.

The complete algorithm for solving the problem of supporting actions for a goal works as follows. The algorithm use the standard backtracking based scheme as it is presented in algorithms 3.1 and 3.10. The difference is that at each decision point (selection of an action into the solution) the described tractability of the remaining problem is checked. If the remaining problem belongs to the tractable class the algorithm switches to the algorithm for solving the tractable problem which is done in backtrack-free manner. If the remaining problem of supports does not belong to the tractable class an action from the clique of action of the highest degree in the clique intersection graph is selected into the solution and the whole solving process continues into the next iteration.

3.4.2 Experimental Evaluation

We evaluated the proposed approach using our experimental implementation written in C++. The integration of the proposed consistency enforcing algorithm into the solving algorithm is similar as that in the case of maintaining arc-consistency and maintaining projection consistency. The algorithm follows the standard GraphPlan algorithm except the part for solving the problem of finding supporting actions for a goal. For this we use maintaining (weak) projection consistency with the heuristic for preferring the tractable case (action from a clique of the highest degree is preferably selected) and when the tractable case is reached we switch to the strong projection consistency as it is described in above paragraphs. Specifically, the tractable case preferring heuristic is used for value selection ordering (selection of the action). For variable ordering we use the standard first fail (smallest domain) heuristic.

We used the same set of planning problems as in the previous sections for the experiments. This allows a direct comparison of performance of all the proposed method. The set of planning problems consists of several instances of various difficulties of Dock Worker Robots, Towers of Hanoi and Refueling Planes planning domain.

We compared the proposed method for solving the tractable case of the problem with the standard GraphPlan, with the *variant C* which maintains arc-consistency and with the version which maintains pure projection consistency. The experiments were again run on the same machine (two AMD Opteron 242 processors - 1600 MHz, with 1GB of memory under Mandriva Linux 10.2). The code was again compiled by the gcc compiler 3.4.3 with maximum optimization for the machine (-O3 -mtune=opteron).

The comparison of the overall solving time (planning graph building time + time of plan extraction phases) of the proposed method based on tractable case with the standard GraphPlan, maintaining arc-consistency method - *variant C*, and the method using projection consistency is shown in figure 3.13. The comparison of times spent in plan extraction phases is shown in figure 3.14. Again the same methods as in figure 3.13 were compared. The comparison of the number of backtracks is shown in figure 3.15. In all the figures 3.13, 3.14, and 3.15 problems along the horizontal axis are ordered according to the ascending solving time of the method with maintaining projection consistency.

The improvement in overall problem solving time is up to 200% compared to the version which uses projection consistency. The improvement in plan extraction time is up to 1000% . The improvement in the number of backtracks is also substantive. Even some problems were solved without backtracking. The improvements are better for problems with more interacting objects and higher action parallelism (for example dwr05). On the other hand there is almost no improvement on problems with no action parallelism (for example han07), which is expectable.

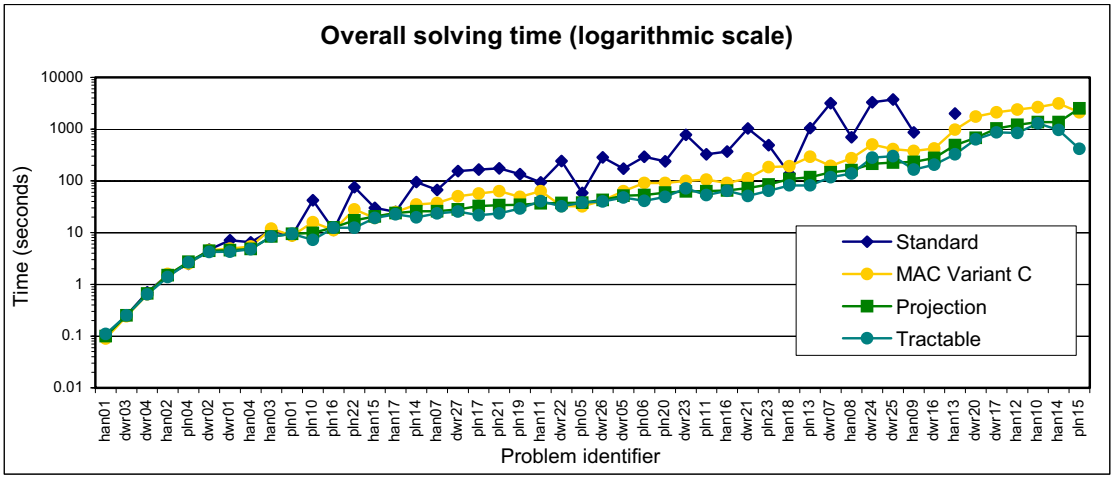


Figure 3.13. *COMPARISON OF OVERALL SOLVING TIMES (LOGARITHMIC SCALE) - (STD, VARC, PRJ, TRACT).* Comparison of the overall solving time of the standard GraphPlan algorithm and enhanced versions which use maintaining arc-consistency of variant C, maintaining projection consistency, and maintaining projection consistency with preference of tractable case for solving the problem of finding supports. Problems on the horizontal axis are listed in the ascending order according to the solving time consumed by the maintaining projection consistency algorithm. Time limit of 1 hour for each problem is used.

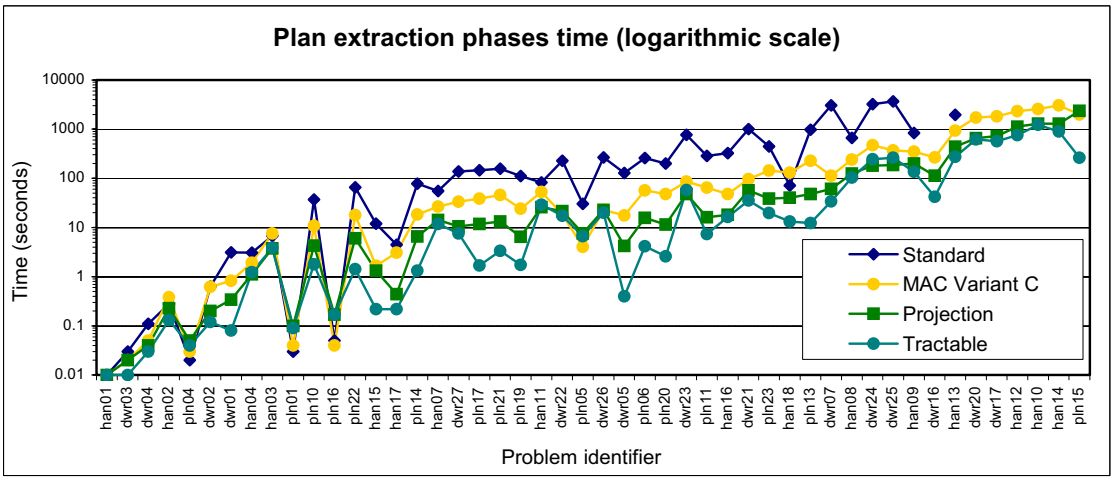


Figure 3.14. *COMPARISON OF PLAN EXTRACTION PHASE TIMES (LOGARITHMIC SCALE) - (STD, VARC, PRJ, TRACT).* Comparison of time spent in plan extraction phases of the standard GraphPlan algorithm and enhanced versions which use maintaining arc-consistency of variant C, maintaining projection consistency, and maintaining projection consistency with preference of tractable case for solving the problem of finding supports. Problems on the horizontal axis are listed in the ascending order according to the solving time consumed by the pure maintaining projection consistency algorithm.

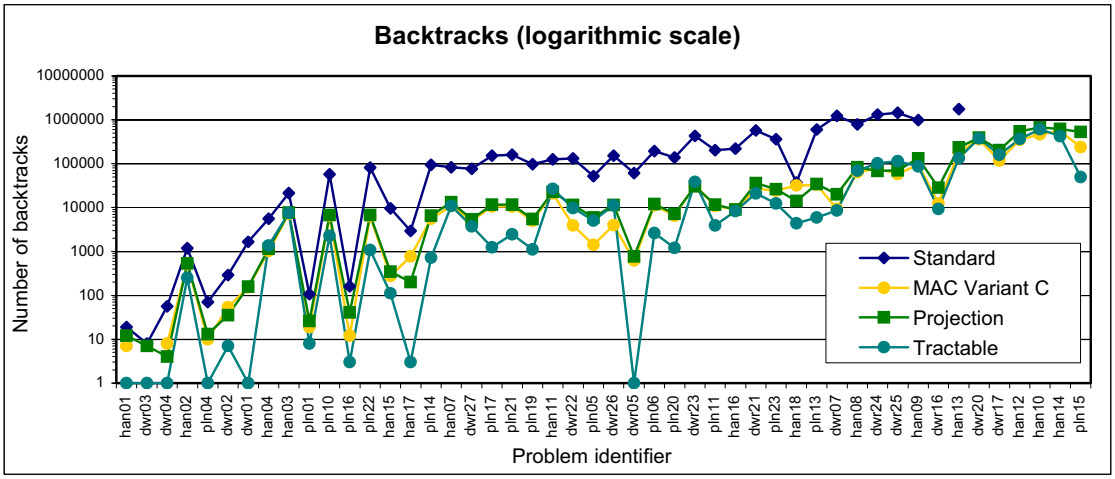


Figure 3.15. *COMPARISON OF NUMBER OF BACKTRACKS (LOGARITHMIC SCALE) - (STD, VARC, PRJ, TRACT).* Comparison of time spent in plan extraction phases of the standard GraphPlan algorithm and enhanced versions which use maintaining arc-consistency of variant C, maintaining projection consistency, and maintaining projection consistency with preference of tractable case for solving the problem of finding supports. Problems on the horizontal axis are listed in the ascending order according to the solving time consumed by the pure maintaining projection consistency algorithm.

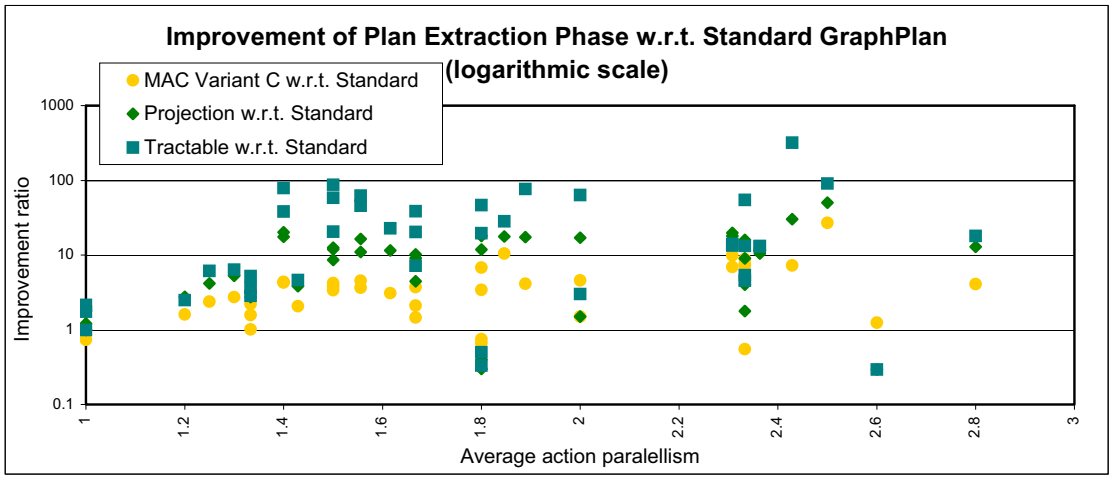


Figure 3.16. *COMPARISON OF IMPROVEMENTS WITH RESPECT TO STANDARD GRAPHPLAN - (VARC, PRJ, TRACT).* Comparison of the standard GraphPlan and enhanced versions based on constraint model with maintaining arc-consistency of variant C and projection consistency for solving the problems of finding supports in terms of improvement ratio of the plan extraction phase depending on the average action parallelism (number of actions in the plan divided by the length of the resulting concurrent plan). Improvements are computed with respect to the standard GraphPlan (which has the ratio 1).

We evaluated the hypothesis that the benefit of preference of the proposed tractable class and the associated specialized algorithm is better on problems with higher action parallelism experimentally. The results of this evaluation are shown in figure 3.16 - improvement ratio of the time of plan extraction phase is shown here. The figure 3.16 shows the improvements of the advanced methods for solving the problem of finding supporting actions with respect to the standard GraphPlan.

The results show that the improvement gained by using the tractable class of the problem of finding supporting actions is more significant for problems higher than zero action parallelism. However, the action parallelism seems not to be the only factor influencing the possible gain of using the tractable class.

3.4.3 Discussion of Results

We described the tractable class of the supports problem using the projection global consistency. Our experiments showed that this class is also useful for practical solving of planning problems since problems of this class arises (with some help) frequently. Using the projection consistency the time spent by solving the problem of finding supports is no more a limiting factor of the planning algorithm. The limiting factor is rather the time spent by building planning graphs and by search across the layers of the planning graph. We consider that it would be interesting to reformulate planning graphs in order to be friendlier to the backward search.

3.5 Difficult Planning Problems

Although our experimental implementation of the enhancements of the GraphPlan algorithm is not designed to compete with today's state-of-the-art planning systems there are planning problems on which our approach is competitive despite its not well optimized implementation.

In previous sections we showed that plan extraction enhanced by the integration of the algorithm for the tractable class improves the plan extraction process significantly. Moreover, we found that our approach is especially successful on difficult problems which force the planner to really perform search to find the solution or to prove that there is no solution (Urquhart, 1987). Such problems encapsulates for example an instance of the Dirichlet's box principle (place $n+1$ pigeons into holes n so that no two pigeons are placed in the same hole). Although these problems are short in length of the input, they are hard to be answered. The solver does not see the principle encoded in the problem formulation. This property of the box principle makes the solver to perform exhaustive search to prove that

the problem has no solution. The algorithm for the problem of finding supports of the tractable class has an advantage in such situation. It can detect insolvability of some subproblems quickly in polynomial time and can prune large parts of the search space in this way.

The performed competitive comparison of our experimental planning system implementing the algorithm for the tractable class with several state-of-the-art planners on the mentioned difficult problems showed surprising results. We chose several planners participating in the International Planning Competition (*IPC*) (Gerevini *et al.*, 2006) for our experimental evaluation. Although it was not our goal to compete with planners from the *IPC* by our experimental planner, the result was that our experimental planner performs better than some of the winners of the *IPC* on selected problems. Moreover, one of the most successful planners in *IPC* - SGPlan 5 (Hsu *et al.*, 2006, 2007) - solved several unsolvable problems (which indicates that the planner is not correct). Another successful planner in *IPC* - *SATPlan* (Kautz *et al.*, 2006, 2007) - seems to be unable to prove nonexistence of the solution.

On the other hand, on the standard benchmark problems our experimental planning system performs worse than planners from *IPC*. Nevertheless, this is expected because our implementation is not so well optimized and does not use any domain specific heuristic. Moreover, it seems that many of the standard benchmark problems can be solved by guessing the solution by some kind of a heuristic mechanism without search. This is allowed by the fact that a solution represents something that may be called a witness. Having the witness for existence of the solution the planning task is finished. In contrast, this approach cannot be used on the mentioned difficult problems. There is no such witness giving evidence that the underlying box principle has no solution. Therefore exhaustive search must be performed.

3.5.1 Experiments

For the competitive evaluation we used several problems encoding (insolvable) Dirichlet's box principle. The set of testing problems is described in appendix A. We compared our version of the GraphPlan algorithm that prefers tractable class with several state-of-the-art planners. The planners were selected according to results of the last two *IPCs* and according to their availability. We were trying to evaluate our approach with respect to the winning planners. However not all planners participating in the *IPC* are available. Finally we selected optimal planners *MaxPlan* (Zhao *et al.*, 2006, 2007), *SATPlan* (Kautz *et al.*, 2006, 2007), *CPT 1.0* (Vidal and Geffner, 2006, 2007), and *IPP 4.1* (Koehler *et al.*, 1997, Koehler, 2007) and satisfying planners *SGPlan 5.1* (Hsu *et al.*, 2006, 2007) and *LPG-td 1.0* (Gerevini and Serina, 2002, 2007). The results are shown in tables 3.6 and 3.7. All the tests were performed on the same testing machine (two AMD Opteron 242 processors - 1600

MHz, with 1GB of memory under Mandriva Linux 10.2). Where source code of the planner was available the system was newly compiled by gcc 3.4.3 with the maximum optimization compilation options for the testing machine (-O3 -mtune=opteron). Otherwise provided executable was used.

Instance	Solvable	SGPlan 5.1 (seconds)	IPP 4.1 (seconds)	MaxPlan/ miniSat 2 (seconds)	SATPlan/ Siege 4 (seconds)	CPT 1.0 (seconds)	LPG-td 1.0 (seconds)
ujam-02_01	no	N/A	**0.00**	**+ 0.00**	∞	0.06	**+ 0.00**
ujam-03_02	no	↓ 0.01	**0.01**	**+ 0.02**	∞	∞	**+ 5.00**
ujam-04_03	no	↓ 0.01	**0.64**	**+ 1.03**	∞	∞	**+ 6.00**
ujam-05_04	no	↓ 0.02	**83.63**	**+ 8.58**	∞	∞	**+ 9.00**
ujam-06_05	no	↓ 0.02	> 600	> 600	∞	∞	> 600
ujam-07_06	no	↓ 0.04	> 600	> 600	∞	∞	> 600
ujam-08_07	no	↓ 0.06	> 600	> 600	∞	∞	> 600
ujam-09_08	no	↓ 0.10	> 600	> 600	∞	∞	> 600
ujam-10_09	no	↓ 0.16	> 600	> 600	∞	∞	> 600
jam-02_01	yes	↑ 0.00	**0.00**	**0.26**	**0.17**	**0.03**	↑ 0.02
jam-03_02	yes	↑ 0.00	**0.00**	**0.25**	**0.16**	**0.04**	↑ 0.01
jam-04_03	yes	↑ 0.00	**0.02**	**0.44**	**0.17**	**0.17**	↑ 0.02
jam-05_04	yes	↑ 0.00	**0.65**	**1.10**	**0.23**	**5.03**	↑ 0.02
jam-06_05	yes	↑ 0.01	**25.8**	**2.77**	**0.92**	**228.75**	↑ 0.01
jam-07_06	yes	↑ 0.01	> 600	**30.92**	**3.01**	> 600	↑ 0.02
jam-08_07	yes	↑ 0.01	> 600	**228.01**	**14.67**	> 600	↑ 0.02
jam-09_08	yes	↑ 0.01	> 600	> 600	**152.01**	> 600	↑ 0.03
jam-10_09	yes	↑ 0.01	> 600	> 600	> 600	> 600	↑ 0.02
holes-02_01	no	↓ 0.00	**0.00**	**+ 0.00**	∞	∞	**0.00**
holes-03_02	no	↓ 0.00	**0.00**	**+ 0.01**	∞	∞	**4.00**
holes-04_03	no	↓ 0.00	**0.00**	**+ 1.04**	∞	∞	**5.00**
holes-05_04	no	↓ 0.00	**0.01**	**+ 15.00**	∞	∞	**5.00**
holes-06_05	no	↓ 0.01	**0.12**	**+ 270.00**	∞	∞	**5.00**
holes-07_06	no	↓ 0.01	**1.89**	> 600	∞	∞	**6.00**
holes-08_07	no	↓ 0.01	**30.34**	> 600	∞	∞	**14.00**
holes-09_08	no	↓ 0.02	**574.60**	> 600	∞	∞	> 600
holes-10_09	no	↓ 0.02	> 600	> 600	∞	∞	> 600

Table 3.6. PERFORMANCE COMPARISON OF PLANNERS ON DIFFICULT PROBLEMS - PART I. Performance of several selected state-of-the-art planners on hard problems encoding Dirichlet's box principle. Overall solving time in seconds is shown. The symbol ↓ indicates that a planner incorrectly solved an insolvable problem, the symbol ↑ indicates non-optimal solution, the symbol + indicates possible inaccuracy (in order of tenths of second) of the measurement, the symbol ∞ indicates divergence of planning process (possible infinite loop). Timeout was 10 minutes for all planners and problems.

The improvement obtained by using tractable class preference algorithm is in order of magnitude compared to the selected state-of-the-art planners on the selected problems. Experiments showed that current planners do not cope well on the selected insolvable problems. Even some planners seems to be falling into an infinite loop (*SATPlan* and *CPT*), another planner incorrectly solved insolvable problems (*SGPlan*). We consider this property as a major obstacle for practical usage of these planners (a user does not get the answer whether the problem can be solved). Although the selected class of testing problems is

rather limited, it represents an important class of difficult problems (Urquhart, 1987) which intrinsically require search to be answered (the solution cannot be guessed).

Instance	Our planner (seconds)	Speedup ratio w.r.t SGPlan 5.1	Speedup ratio w.r.t IPP 4.1	Speedup ratio w.r.t. MaxPlan	Speedup ratio w.r.t. SATPlan	Speedup ratio w.r.t. CPT	Speedup ratio w.r.t. LPG-td
ujam-02_01	0.06	N/A	N/A	N/A	N/A	1.00	N/A
ujam-03_02	0.54	N/A	0.02	+ 0.04	N/A	N/A	*+ 9.25*
ujam-04_03	3.39	N/A	0.19	+ 0.30	N/A	N/A	*+ 1.76*
ujam-05_04	23.88	N/A	*3.50*	+ 0.35	N/A	N/A	+ 0.37
ujam-06_05	177.9	N/A	*> 3.37*	*> 3.37*	N/A	N/A	*> 3.37*
ujam-07_06	> 600	N/A	N/A	N/A	N/A	N/A	N/A
ujam-08_07	> 600	N/A	N/A	N/A	N/A	N/A	N/A
ujam-09_08	> 600	N/A	N/A	N/A	N/A	N/A	N/A
ujam-10_09	> 600	N/A	N/A	N/A	N/A	N/A	N/A
jam-02_01	0.03	N/A	N/A	*8.66*	*5.66*	1.00	N/A
jam-03_02	0.09	N/A	N/A	*2.77*	*1.77*	0.44	N/A
jam-04_03	0.25	N/A	0.08	*1.76*	0.68	0.68	N/A
jam-05_04	0.60	N/A	*1.08*	*1.80*	0.38	*8.38*	N/A
jam-06_05	1.29	N/A	*20.00*	*2.14*	0.71	*177.32*	N/A
jam-07_06	2.48	N/A	*> 241.93*	*12.46*	*1.21*	*> 241.93*	N/A
jam-08_07	4.60	N/A	*> 130.43*	*49.56*	*3.19*	*> 130.43*	N/A
jam-09_08	8.77	N/A	*> 68.42*	*> 68.42*	*17.33*	*> 68.42*	N/A
jam-10_09	17.05	N/A	*> 35.19*	*> 35.19*	*> 35.19*	*> 35.19*	N/A
holes-02_01	0.00	N/A	N/A	N/A	N/A	N/A	N/A
holes-03_02	0.02	N/A	N/A	+ 0.50	N/A	N/A	*200.00*
holes-04_03	0.04	N/A	N/A	*+ 26.00*	N/A	N/A	*125.00*
holes-05_04	0.12	N/A	0.08	*+ 125.00*	N/A	N/A	*41.66*
holes-06_05	0.27	N/A	0.44	*+ 1000.00*	N/A	N/A	*18.51*
holes-07_06	0.55	N/A	*3.44*	*> 1090.00*	N/A	N/A	*10.90*
holes-08_07	1.08	N/A	*28.09*	*> 555.55*	N/A	N/A	*12.96*
holes-09_08	2.08	N/A	*276.25*	*> 288.46*	N/A	N/A	*> 288.46*
holes-10_09	4.06	N/A	*> 147.78*	*> 147.78*	N/A	N/A	*> 147.78*

Table 3.7. PERFORMANCE COMPARISON OF PLANNERS ON DIFFICULT PROBLEMS - PART II. Comparison of selected state-of-the-art planner with our experimental planning system on several hard planning problems encoding Dirichlet's box principle. The symbol N/A indicates that a comparison cannot be made.

In our minor experiments we also measured memory consumed by planners. While our experimental planner together with *SGPlan* fit below 16MB of memory, other tested planners consume more than 100MB of memory on tested problems (approximate consumption was *IPP* - 300MB, *MaxPlan* - 600MB, *SATPlan* - 400MB, *CPT* - 150MB, *LPG-td* - 700MB - even on larger problems exceeded the whole 1GB of the testing machine). Next we did some experiments on comparison of pure performance of planners. We found that our experimental implementation is quite uncompetitive since for example *IPP* is able to examine approximately 300000 actions per second while our implementation can examine no more than 1000 actions per second.

These results show that the algorithm for solving the tractable class is competitive to the state-of-the-art planners on a certain set of difficult problems. The result is especially interesting due to the fact that we did not use any heuristics or optimization to further in-

crease the performance. Our method was implemented exactly according to the described symbolic codes and propositions.

The results show that structural properties of the solved problems play an important role and if it is possible to uncover them and utilize them in the solving algorithm the performance benefit may be significant. We particularly showed this in case of projection consistency which is based on the structures of complete graphs encoded in the problem formulation.

3.6 Summary and Conclusion

We were dealing with solving planning problems using planning graphs in this chapter. We discovered certain inefficiencies of the standard algorithm for solving planning problems over planning graphs - GraphPlan. Particularly we found that the standard GraphPlan solves inefficiently the problem of finding supporting actions for a certain goal. We proposed several methods how to improve the solving of this problem of supports.

First, we proposed a method based on maintaining arc-consistency. We designed a special constraint model for modeling the problem of finding supporting actions. We proposed several variants of maintaining arc-consistency in the model. The performed experimental evaluation of the usage of constraint model with maintaining arc-consistency with the standard GraphPlan algorithm showed that maintaining arc-consistency improved the solving process significantly in terms of overall solving time.

The overall improvement obtained by the application of arc-consistency inspired us to develop stronger consistency specially designed for the solved problem. The principal observation is that arc-consistency is a local method only (that is only the variables neighboring closely through constraints interact). So, we were interested in a more global method which eventually takes into account structural properties of the problem.

According to these guidelines we proposed a consistency method which we called projection consistency. The projection consistency is a consistency technique specially designed to enforce certain level of consistency within the problem of finding supporting actions for a goal. This type of consistency is global, that is it takes into account the whole problem, and it exploits the structural properties of the problem. As the useful structural properties the technique exploits the sets of actions that form complete sub-graphs in the graph of mutually excluded actions (the set of actions form a clique of mutexes). The important property of the set of actions from a single clique is that at most one of these actions can be selected into the solution. This property allows us to introduce special counting that are used to determine whether a certain action contributes enough to the solution (to the goal) and if this is not the case the action can be ruled out. This allows us to reduce the search space significantly.

Our experimental evaluation showed that projection consistency performs yet better than arc-consistency of all the proposed propagation variants.

The final result in this chapter is a so called tractable class of the problem of finding supporting actions. We found that projection consistency can be used to solve certain instances of the problem of supports completely without search. We defined the tractable class of the problem of supports and we proposed a heuristic which transforms the general problem of supports to the problem belonging into the tractable class.

We again performed experiments devoted to the comparison of all the proposed methods. The result is that the version in which tractable class of the problem was preferred and solved by the specialized algorithm performs best. Some of the planning problems were even solved without backtracking.

Finally, we performed competitive comparison of our method based on the tractable class of the problem of supports with today's state-of-the-art planners on a set of difficult planning problems. The performed experiments showed that our method is better in terms of overall solving time than the tested planners on selected difficult problem. However, the set of problems in this final experiment comprise only a limited class of problems which does not allow us to state that our method based on tractable class is better generally.

CHAPTER 4

CONTRIBUTIONS TO BOOLEAN SATISFIABILITY

In the previous chapter we studied the problem of finding supporting actions for a sub-goal in artificial intelligence planning context. This is some kind of an important sub-problem which must be solved many times when solving AI planning problems using the *planning graphs* (Blum and Furst, 1997). It was shown in the previous chapter that the problem of finding supporting actions is *NP*-complete. In doing so a conversion of an instance of the SAT problem to the instance of the problem of finding supports was used. This proof uncovered some interesting similarities between the SAT problem and the supports problem. The positive experience made with the method on planning problems and the observed similarity lead us to the idea of adapting the technique of the greedy clique decomposition to solve SAT problems.

Results presented in this chapter were published in (Surynek, 2007f, 2007g). We adapt here the previously developed projection consistency for Boolean satisfaction. To distinguish between the original projection consistency and its new adaptation we call the new concept *clique consistency*.

Boolean formula satisfaction problems and SAT solving techniques play an extremely important role in theoretical computer science as well as in practice. The question of whether there exist a complete polynomial time SAT solver is a key question for theoretical computer science and is open for many years (the *P vs. NP problem* - Cook, 1971). On the other hand the practical use of SAT problems and SAT solvers in real life applications is also very intensive. Applications of SAT solving techniques range from microprocessor verification (Velev and Bryant, 2003) and field-programmable gate array design (Nam *et al.*, 2002) to solving AI planning problems by translating them into Boolean formulas (Kautz and Selman, 1992).

An excellent performance breakthrough was done in solving SAT problems over the past years. Thanks to new algorithms and implementation techniques focused on real life SAT problems many of the today's benchmark problems (Le Berre and Simon, 2005; Sinz,

2006) are solved by state-of-the-art solvers (Eén and Sörensson, 2005, 2007; Fu et al., 2007; Gershman and Strichman, 2007; Moskewicz *et al.*, 2001; Gershman and Strichman, 2005) in time proportional to the size of the input. It seems that the difficulty of many SAT benchmark problems consists in their size only. A lot of smaller benchmark problems are solved in real-time by today's state-of-the-art SAT solvers. The observation that can be deduced upon these facts is that there is almost no chance to compete with the best SAT solvers by a newly written SAT solver on these problems. That is why we are concentrating on difficult instances of SAT problems only, where the word difficult means difficult for today's state-of-the-art SAT solvers.

A valuable set of difficult (in the mentioned sense) problems was collected by Aloul (2007). Although these problems are small in the length of the input formula they are difficult to be answered. The detailed discussion about hardness of these problems is given in (Aloul *et al.*, 2002). One of the aspects of problem difficulty is that these problems are mostly unsatisfiable (and this fact is well hidden in the problem). The solver cannot guess a solution using its advanced techniques and heuristics in such a case and it must really perform some search in order to prove that there is no solution. In the case of a positive answer the satisfying valuation of variables serves a witness (of small size) certifying the existence of at least one solution. If the solver obtains (possibly by guessing) a witness its task is finished. In contrast to this, there is no such small witness in the case of a negative answer so the search must be performed.

Our contribution to solving SAT problems consists of preprocessing and reformulating the input Boolean formula in the *CNF* (*conjunctive normal form* - conjunction of disjunctions). The result of this processing is either the answer whether the input formula is unsatisfiable or a new formula (hopefully simpler) with the same set of satisfying valuations as that of the input one. If the input formula is not decided by the preprocessing phase then the preprocessed formula is sent to the SAT solver of the user's choice. The idea behind this process is to make the task for the SAT solver easier by deciding the input formula within the fast preprocessing phase or by providing an equivalent but simpler formula to the SAT solver. Experiments showed that the solving process over the above mentioned difficult SAT benchmarks speeds up by the order of magnitude after using our approach.

The reformulation within the preprocessing phase itself is simple. We are viewing the input Boolean formula in a CNF as a graph. For each *literal* (variable or its negation) of the input formula we consider a vertex and for each conflict between literals we consider an edge. Conflicting literals are those that cannot be both satisfied in a single valuation of variables, for example positive and negative literals of the same variable are conflicting. Generally, a set of literals of a formula is conflicting if the formula entails that at most one of the literals can be *true* . To be able to use our reasoning based on the clique decomposition we need a graph with appropriately large *complete sub-graphs* (cliques). That is, we need some kind of a good approximation of the sets of conflicting literals. Unfortunately the graph arising from the above interpretation of the Boolean formula in *CNF* is rather

sparse (the largest clique is of size 2). That is why we apply further inference by which we deduce more conflicts between the literals and which allow us to introduce more edges into the graph. We are using singleton arc-consistency (Bessière and Debruyne, 2005) as the inference technique for deducing new edges.

Having the graph constructed from the input CNF formula, a clique decomposition of this graph is found by a greedy algorithm (we do not need an optimal clique decomposition; we need just some of the reasonable quality). The important property of the constructed clique decomposition is that at most one literal from each clique can be assigned the value *true*. In this situation we perform certain kind of literal contribution counting to rule out the literals that can never be *true*. To do this, the maximum number of satisfied clauses by literals of each clique is calculated. Then a literal of a certain clique can be ruled out if the literals from the other cliques together with the selected literal do not satisfy enough clauses to satisfy the input formula.

Although this problem reformulation seems weak it provides strong reasoning about the dependencies among clauses of the CNF Boolean formula and about the effect of the selection of a value for a variable on the overall satisfiability of the formula. Moreover if all the literals are ruled out during the preprocessing phase, the input formula is obviously unsatisfiable. Experimental evaluation showed that this happen frequently on difficult SAT problems. For other cases a new formula in the CNF equivalent to the input formula is produced. The new formula is constructed from the original one by adding clauses that capture all the dependencies inferred by the initial singleton arc-consistency stage and by the literal contribution counting based on the clique decomposition.

The chapter is organized as follows. A detailed formal description of the reformulation of a SAT instance using the greedy clique decomposition is given in section 4.1. The subsequent section 4.2 is devoted to some experimental comparison of our approach with the existing state-of-the-art SAT solvers. We are discussing the contribution of our method within this section too. Finally we put our work in relation to similar works in the field of Boolean satisfiability and we propose some future research directions of the studied topic.

4.1 SAT Reformulation Using Clique Decomposition

We formally describe the details of the process of SAT problem reformulation in this section. Let $B = \wedge_{i=1}^{n} \vee_{j=1}^{m_i} x_j^i$ be the input Boolean formula in CNF where x_j^i is a literal (variable or its negation) for all possible i and j. A sub-formula $\vee_{j=1}^{m_i} x_j^i$ of the input formula B for every possible i is called a clause. The ith clause of the formula B will be denoted as C_i in the following paragraphs. As it was mentioned in the introduction, the basic idea of the SAT problem reformulation consists in viewing the input formula as an undirected graph in which the internal structure of the formula is captured in some way. To be more

specific, the graph will capture the pairs of conflicting literals and it will be constructed in several stages as the following section shows.

4.1.1 Inference of Conflicting Literals

Let us start by the construction of an undirected graph $G_B^1 = (V_B^1, E_B^1)$ which will represent trivially conflicting literals in the given *CNF* formula B. The graph will be called a *graph of trivial conflicts*. The graph G_B^1 will then undergo some further inference process by which the additional conflicts will be inferred. We will denote the resulting undirected graph as $G_B^2 = (V_B^2, E_B^2)$ and call it an *intermediate graph of conflicts*.

The construction of the undirected graph G_B^1 is simple. A vertex is introduced into the graph G_B^1 for each literal occurring in the formula B, that is $V_B^1 = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m_i} \{x_j^i\}$ (notice that $|V_B^1|$ is typically smaller than the length of the formula, since literals may occur many times in the formula while only once in the graph). The construction of the set of edges E_B^1 is also straightforward. An edge $\{x_j^i, x_l^k\}$ is introduced into the graph G_B^1 if the literals x_j^i and x_l^k are trivially conflicting, that is if one is a variable v and the other is $\neg v$ for some Boolean variable v. The graph G_B^1 is completed by performing the above step for all possible pairs of conflicting literals. The interpretation of the graph of conflicts is that if a literal corresponding to a vertex is selected to be assigned the value *true* all literals corresponding to the neighboring vertices must be assigned the value *false*.

An example graph resulting from the described process over a selected benchmark problem is shown in the left part of figure 4.1. The resulting graph is visibly sparse, since there are edges only between the literals of the same variable. Hence it is not a good starting point for our method and a further inference mechanism for discovering more conflicting pairs of literals (more edges for the graph) must be applied. This further inference mechanism takes the already constructed graph G_B^1 and augments it by adding new edges. The result of this stage is an intermediate graph of conflicts G_B^2.

The process of construction of graph G_B^2 exploits techniques known from standard SAT resolution approaches and from *constraint programming* (Dechter, 2003) - *unit propagation* (Dowling and Gallier, 1984; Zhang and Stickel, 1996), *arc-consistency* (*AC*) (Mackworth, 1977) and *singleton arc-consistency* (*SAC*) (Bessière and Debruyne, 2005). Before describing the construction of the graph G_B^2 let us recall these notions. We modify the above concepts slightly for the SAT domain to prepare them for our purposes. The following definitions assume the input formula B in CNF and a corresponding graph of conflicts G_B (for example the graph G_B^1 expressing the trivial conflicts).

Definition 4.1 (ARC-CONSISTENCY IN SAT INSTANCE W.R.T. THE GRAPH OF CONFLICTS). Consider two clauses C_i and C_k for $i,k \in \{1,2,\ldots,n\}$, $i \neq k$ of the formula B. A literal x_j^i ($j \in \{1,2,\ldots,m_i\}$) from the clause C_i is *supported* by the clause C_k with respect to the

given graph of conflicts G_B if there exists a literal x_l^k ($l \in \{1,2,\ldots,m_k\}$) from the clause C_k, such that the literals x_j^i and x_l^k are not in a conflict with respect to the graph G_B (not connected by an edge). An ordered pair of clauses (C_i, C_k) of the formula B is called an *arc* in this context. An arc (C_i, C_k) for some $i,k \in \{1,2,\ldots,n\}$ is *consistent* (*or arc-consistent*) with respect to the graph of conflicts G_B if all the literals of the clause C_i are supported by the clause C_k with respect to the graph of conflicts G_B. The formula B is called *arc-consistent* with respect to the graph of conflicts G_B if all the arcs (C_i, C_k) for all $i,k = 1,2,\ldots,n$ are arc-consistent with respect to the graph of conflicts G_B. □

Let us note that our definition is based on a dual view of the satisfaction problem. That is, we use the clauses of the formula as the CSP variables (Dechter, 2003) instead of the original Boolean variables. Having these CSP variables, (CSP) constraints necessary for the definition of arc-consistency arise naturally.

The reason for the definition of arc-consistency is that the literals which are not supported according to the definition cannot be assigned the value *true* (this means that the corresponding variable cannot be assigned the value *false* in the case of a negative literal). So the solver can rule out such literals from further attempts to assign them the value *true*, which may reduce the size of the search space. Notice that the definition has the graph of conflicts G_B as a parameter. It is possible to put any correct graph of conflicts as a parameter of this definition, whereas correct means, that if $\{y,z\}$ is the edge of the graph then $B \Rightarrow y \neq z$ must be a tautology. This is obviously true for the graph of trivial conflicts G_B^1. Notice also that if we use the graph of trivial conflicts G_B^1 the definition becomes identical to unit propagation (Dowling and Gallier, 1984; Zhang and Stickel, 1996).

Having the Boolean formula B the question is how to make it arc-consistent with respect to the given graph of conflicts. For this purpose we adopt techniques developed in constraint programming and by SAT community, namely the arc-consistency enforcing algorithms (Dechter, 2003; Mackworth, 1977) and unit propagation (Dowling and Gallier, 1984; Zhang and Stickel, 1996**)**. There is a great variety of such algorithms; however their common feature is the search for supports for every value (literal) which is suspected of not being supported. The main difference among these algorithms is the efficiency of the search for supports. If an unsupported literal is detected it is ruled out. Ruling out an unsupported literal may cause that some other literal loses its only support. This chain-like propagation of changes continues until a stable state is reached. For purposes of the SAT domain this propagation process is usually augmented by an additional simplification rule. If the consistency enforcing algorithm detects that within some clause there is only one literal that can be selected to be *true*, it is fixed to value *true* and the corresponding clause is cut out from further reasoning (this is exactly the simplification rule from unit propagation).

Unfortunately, the defined arc-consistency over Boolean formulas in the CNF form is too weak to infer significantly more conflicts than those already present in the graph of trivial conflicts. Therefore we need to make the consistency stronger. Perhaps the simplest way

to do this is to make the selected consistency technique *singleton* (Bessière and Debruyne, 2005). The following definition again assumes the Boolean formula B and the corresponding graph of conflicts G_B (again the graph of trivial conflicts G_B^1 can be used).

Definition 4.2 (SINGLETON ARC-CONSISTENCY IN A SAT INSTANCE W.R.T. THE GRAPH OF CONFLICTS). A literal x_l^k ($l \in \{1,2,\ldots,m_k\}$) from a clause C_k for $k \in \{1,2,\ldots,n\}$ of the formula B is *singleton arc-consistent* with respect to the given graph of conflicts G_B if the formula obtained from B by replacing the clause C_k by the literal x_l^k (the resulting formula is $(\wedge_{i=1}^{k-1} \vee_{j=1}^{m_i} x_j^i) \wedge x_l^k \wedge (\wedge_{i=k+1}^{n} \vee_{j=1}^{m_i} x_j^i)$) is arc-consistent with respect to the graph of conflicts G_B. □

Unsupported literals in the formula modified by replacing the clause C_k by the literal x_l^k are in conflict with the literal x_l^k. This is quite intuitive, the selection of the literal x_l^k to be assigned the value *true* rules out some other literals. Hence these literals are in conflict with the selected literal x_l^k. Having singleton arc-consistency we are ready to infer new edges for the graph of conflicts.

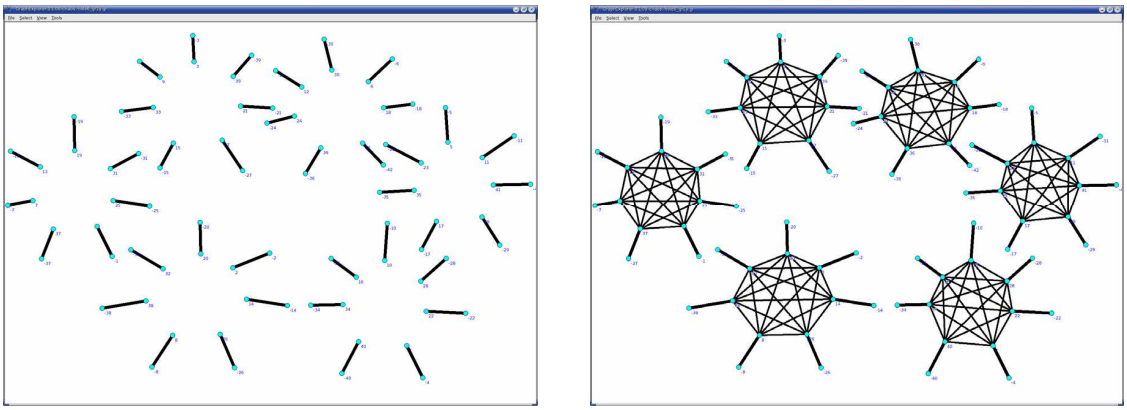


Figure 4.1. GRAPH OF TRIVIAL CONFLICTS AND INTERMEDIATE GRAPH OF CONFLICTS. The left part of the figure shows a graph of trivial conflicts for the SAT benchmark problem pigeon-hole principle number 6 (hole06.cnf). Vertices represents literals, edges are between pairs of positive and negative literals of the same variable. The right part of the figure shows an intermediate graph of conflicts inferred from the original graph of the left by singleton arc-consistency. The graph contains edges from the original graph plus the inferred edges. Six non-trivial cliques each containing seven vertices are clearly visible and can be found by a simple greedy algorithm.

The intermediate graph of conflicts G_B^2 is constructed from the graph of trivial conflicts G_B^1 in the following way. Initially the graph G_B^2 is identical to the graph G_B^1, that is we start with the initialization $V_B^2 \leftarrow V_B^1$ and $E_B^2 \leftarrow E_B^1$. Then for every literal $y \in V_B^2$ singleton arc-consistency with respect to the graph of conflicts G_B^1 is enforced. If the consistency discovers some unsupported literals, say literals $z_1, z_2, \ldots, z_m$, edges $\{y, z_i\}$ for all

$i = 1,2,\ldots,m$ are added into the set of edges E_B^2. Enforcing singleton arc-consistency has the worst case time complexity of $O(|B|k^2m^3)$ if we use AC-3 algorithm as the core of consistency enforcing procedure where m is the maximum size of the clause of the formula (that is, $m = \max_{k=1,2,\ldots,n} m_k$). We need to enforce arc-consistency by the AC-3 algorithm at most $|B|$ times where $|B|$ is the size of the formula B. The AC-3 algorithm itself has the worst case time complexity of $O(k^2m^3)$.

An example of the resulting graph of conflicts is shown in the right part of the figure 4.1. It is constructed from the original graph of trivial conflicts from the left part of the figure 4.1. The cliques in the graph are clearly visible.

The described process of inference of conflicting literals is relatively generic. Both different initial graphs of trivial conflicts as well as consistency techniques different from arc-consistency and singleton arc-consistency for inference of new edges can be used. Both entities, graphs and consistency techniques, may be considered as parameters of the method.

4.1.2 Clique Decomposition and Literal Contribution Counting

To deduce yet more information from the graph of conflicts $G_B^2 = (V_B^2, E_B^2)$ a clique decomposition of the graph is constructed. Formally, a partition of vertices $V_B^2 = K_1 \cup K_2 \cup \ldots \cup K_s$ such that each set of vertices K_i for $i = 1,2,\ldots,s$ induces a clique over the set of edges E_B^2 and $K_i \cap K_j = \emptyset$ for all $i, j = 1,2,\ldots,s \wedge i \neq j$. Let E_{K_i} denotes the set of edges induced by the clique K_i, let E_R denotes the set of edges outside the clique decomposition, that is $E_R = E_B^2 - \bigcup_{i=1}^{s} E_{K_i}$. The inference method based on literal contribution counting performs best if cliques of the decomposition are large. The better the quality of the decomposition is the stronger results are produced by the inference method. However, the problem of finding the optimal clique decomposition with respect to the above criterion is *NP*-complete (Golumbic, 1980). Experiments showed that the simple greedy algorithm can find a clique decomposition of acceptable quality (with respect to clique sizes and the number of edges outside the decomposition - see algorithm 3.5).

The greedy algorithm performed over the graph from the right part of the figure 4.1 finds the clique decomposition consisting of six non-trivial cliques of size seven (there are also trivial cliques consisting of a single vertex). The fact that at most one literal from a clique can be selected to be assigned the value *true* is used in our inference method.

For the following definitions we assume a Boolean formula $B = \wedge_{i=1}^{n} \vee_{j=1}^{m_i} x_j^i$ and the corresponding clique decomposition $V_B^2 = K_1 \cup K_2 \cup \ldots \cup K_s$ of the intermediate graph of conflicts $G_B^2 = (V_B^2, E_B^2)$. Next let $I \subseteq \{1,2,\ldots,n\}$ be a set of indexes of some clauses of the formula B. The set I defines a sub-formula B_I of the formula B, where $B_I = \wedge_{i \in I} C_i$.

Definition 4.3 (LITERAL CONTRIBUTION). A *contribution of a literal* y to the sub-formula B_I is defined as the number of clauses of B_I in which the literal y occurs and it is denoted as $c(y,I)$. □

Definition 4.4 (CLIQUE CONTRIBUTION). A *contribution of a clique* $K \in \{K_1,K_2,\ldots,K_s\}$ to the sub-formula B_I is defined as $\max_{y\in K}(c(y,I))$ and it is denoted as $c(K,I)$. □

The concept of clique contribution is helpful when we are trying to decide whether it is possible to satisfy the sub-formula B_I using the literals from the clique decomposition. If for instance $\sum_{i\in I} c(K_i,I) < |I|$ holds then the sub-formula B_I cannot be satisfied and hence also B cannot be satisfied. Moreover, we can handle a more general case as it is described in the following definitions.

Definition 4.5 (CLIQUE-CONSISTENT LITERAL). A literal $y \in K_i$ for $i \in \{1,2,\ldots,n\}$ is said to be *clique-consistent with respect to the sub-formula* B_I if $\sum_{j\in I, j\neq i} c(K_j,I) \geq |I| - c(y,I)$. □

Definition 4.6 (CLIQUE-CONSISTENT FORMULA). A formula B is *clique-consistent with respect to the sub-formula* B_I if all the literals of the formula B are clique-consistent with respect to B_I. □

A clique-inconsistent literal with respect to some sub-formula of B cannot be selected to be assigned the value *true*. Thus such literals can be ruled out from further reasoning.

Proposition 4.1 *(CORRECTNESS OF CLIQUE-CONSISTENCY). Clique-consistency with respect to a sub-formula* B_I *is correct. That is, the set of solutions of the formula* B *is the same as the set of solutions of the* B' *which we obtain from* B *by enforcing clique-consistency with respect to a sub-formula* B_I. ■

Proof. We show that an inconsistent literal cannot be assigned the value *true*. Let $y \in K_i$ be an inconsistent literal for some $i \in \{1,2,\ldots,s\}$. That is $\sum_{j\in I, j\neq i} c(K_j,I) < |I| - c(y,I)$ holds. After selection of y there remain $|I| - c(y,I)$ clauses in B_I to be satisfied. However, by selecting at most one literal from the remaining cliques we satisfy at most $\sum_{j\in I, j\neq i} c(K_j,I)$ clauses in B_I. ■

The remaining question is how to select the described sub-formulas B_I of B which are used for computation of the clique-inconsistent literals. This selection is crucial for the strength of the proposed clique-consistency. It is clear that we need to rule out as many as possible inconsistent literals. As it is impossible to compute the defined consistency with respect to all such sub-formulas of B, because there are too many sub-formulas, we need to

select a subset of them carefully. We use sub-formulas $B_{I_r} = \wedge_{i \in I_r} C_i$ of B, where $I_r = \{i \in \{1,2,\ldots,n\} \mid m_i = r\}$ for all possible $r \in \mathbb{N}$ for which B_{I_r} is not empty (we suppose that a clause of B does not contain an individual literal more than once). Let us note that we do not know whether this selection is the best possible.

Proposition 4.2 *(COMPLEXITY OF CLIQUE-CONSISTENCY ENFORCING ALGORITHM). There exists a polynomial time algorithm for enforcing clique-consistency with respect to a sub-formula of a given input formula.* ■

Proof. We use a slight adaptation of algorithm 3.6. Its time complexity is $O(|B_I||B|)$ which is $O(|B|^2)$, where $|B|$ denotes the size of the formula. ■

Having such an algorithm it is possible to extend it for multiple sub-formulas B_{I_r} simply by running the algorithm for each $r \in \mathbb{N}$ for which B_{I_r} is non-empty. Since r is linear in size of the input the resulting algorithm is also polynomial.

4.1.3 Output of the Reformulation Process

At this point everything is ready to introduce the final step of our reformulation method. We will be constructing a modified formula β which is initially set to be identical to B. We will further preprocess B by the singleton version of the defined clique-consistency. Conflicts inferred by this further preprocessing will be stored in a new graph of conflicts $G_B^3 = (V_B^3, E_B^3)$ which is initially set to be the same as the graph G_B^2. The graph G_B^3 will be called a *final graph of conflicts* in this context.

Singleton clique-consistency is computed in the following way. For each literal y of the input formula B we enforce clique-consistency for the formula obtained from B by selecting a literal y to be assigned the value *true*. More precisely, clauses containing y are removed and the negation of the literal y is removed from remaining clauses of B (removal of a literal x_k^i from the clause $C_i = \vee_{j=1}^{m_i} x_j^i$ of the formula B is defined as replacement of the clause C_i by the clause $(\vee_{j=1}^{k-1} x_j^i) \vee (\vee_{j=k+1}^{m_i} x_j^i)$). The clique-consistency is then enforced for the resulting formula. Some literals may be found inconsistent during consistency enforcing. These literals are in conflict with the literal y. If for some clause all its literals are found inconsistent with y then the literal y cannot be selected to be *true* and a new clause $\neg y$ is added to β ($\beta \leftarrow \beta \wedge \neg y$). Otherwise the conflicting literals are stored in the graph of conflicts G_B^3 as new edges (that is, if literal y is in conflict with literal z, the edge $\{y,z\}$ is added to G_B^3). The graph of conflicts G_B^3 resulting from processing the intermediate graph of conflicts G_B^2 for the SAT benchmark problem from figure 4.1 is shown in figure 4.2.

If for some clause it is discovered by the clique-consistency that none of its literals can be assigned the value *true* the process terminates with the answer that the formula B cannot be satisfied. This outcome is ensured by the correctness of the method. Our experiments showed that this situation is the most successful case, because an answer to the satisfiability problem is obtained in polynomial time without further expensive search for a solution.

If the process does not terminate with the negative answer then all the edges of the graph of conflicts G_B^3 are translated into new clauses of the formula β. That is, for every edge $\{y,z\} \in E_B^3$ we add a clause $\neg y \vee \neg z$ into the formula β ($\beta \leftarrow \beta \wedge (\neg y \vee \neg z)$) (it is possible to omit edges $\{x,\neg x\} \in E_B^3$). The resulting formula β is equivalent with the original input formula B. Notice that the conflicts inferred by the preceding reformulation stages are also reflected in the formula β, since the graph G_B^3 subsumes the preceding graphs of conflicts G_B^1 and G_B^2. The formula β is finally sent to the SAT solver of the user's choice.

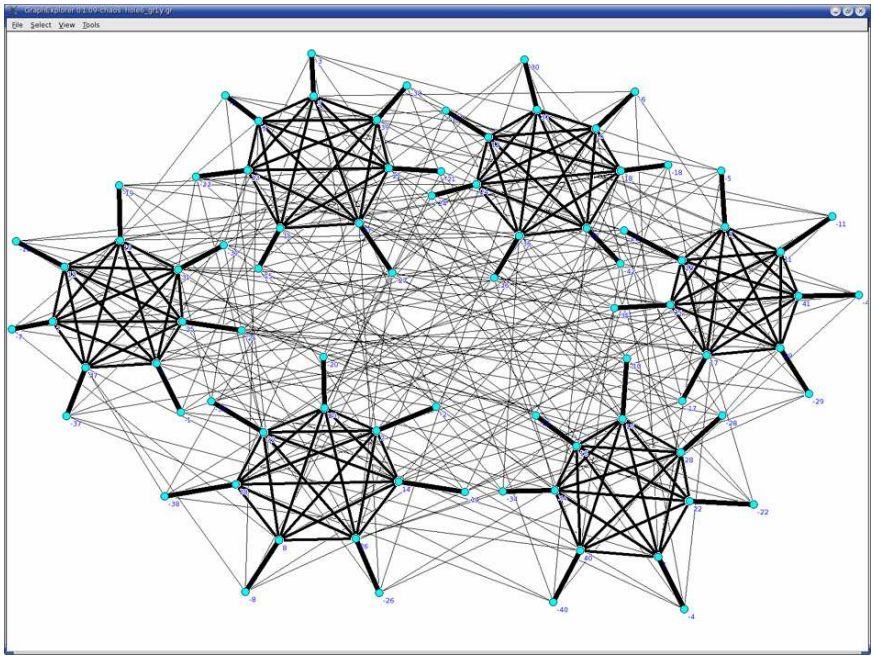


Figure 4.2. FINAL GRAPH OF CONFLICTS. A final graph of conflicts for the SAT benchmark problem pigeon-hole principle number 6 (hole06.cnf). The graph contains edges from the intermediate graph of conflicts from figure 4.1 plus the edges inferred by singleton clique-consistency.

4.2 Experimental Results

We chose three state-of-the-art SAT solvers for comparison with our reformulation method. The SAT solvers of our choice were *zChaff* (Fu *et al.*, 2007; Moskewicz *et al.*, 2001), *Hai-*

faSAT (Gershman and Strichman, 2005, 2007) and *MiniSAT* (a version with *SATElite* pre-processing integrated) - (Eén and Sörensson, 2005, 2007) - (we used the latest available versions to the time of writing this thesis). Our choice was guided by the results of several last SAT competitions - *SAT Competition 2005* and *SAT Race 2006* (Le Berre and Simon, 2005; Sinz, 2006) in which these solvers belonged to the winners. The secondary guidance was that complete source code (in C/C++) for all these solvers is available on web pages of their authors. As we implemented our method in C++ too, this fact allowed us to compile all source codes by the same compiler with the same optimization options which guarantees more equitable conditions for the comparison (a complete source code implementing our method in C++ is available on the attached removable medium). All the tests were run on the machine with two AMD Opteron 242 processors (1600 MHz) with 1GB of memory under Mandriva Linux 10.2. Our method as well as the listed SAT solvers were compiled by the gcc compiler version 3.4.3 with options provided maximum optimization for the target testing machine (-O3 -mtune=opteron). Although the testing machine has two processors no parallel processing was used.

4.2.1 Difficult SAT Instances Selected for Experiments

The testing set consisted of several difficult unsatisfiable SAT instances. This set of benchmark problems was collected by Aloul (Aloul, 2007) and it is provided at his research web page. The details about hardness and construction of these instances are discussed in (Aloul *et al.*, 2002), but let us briefly introduce the problems here too.

Pigeon Holes Instances. *[holes]* This is a standard SAT benchmark encoding the pigeon hole principle problem. The problem asks whether it is possible to place $n+1$ pigeons into n holes without two pigeons being in the same hole. The problem is obviously unsatisfiable. We used seven instances of this problem ranging from 6 to 12 holes. ●

Randomized Urquhart Instances. *[urq]* This set of benchmark problems contains several artificially constructed hard unsatisfiable instances. More details about these problems are provided in (Urquhart, 1987). In addition, the problems were randomized for our testing purposes. We used four instances of the problems of this type. ●

Field Programmable Gate Array Routing Instances. *[fpga, chnl]* This benchmark problem resembles the pigeon holes problem. The question is whether it is possible to route n connections through m tracks provided by the field programmable gate array component. If $n > m$ the problem cannot be satisfied. We used sixteen unsatisfiable instances of this problem for various numbers of required routes and connections. Two different encodings of the

problem are used - denoted *fpga* and *chnl*. More details about the encoding of this problem are provided in (Nam *et al.*, 2002). ●

4.2.2 Effect of Problem Reformulation

For each benchmark SAT instance we measured the overall time necessary to decide its satisfiability. The results are shown in table 4.1 and table 4.2. The speedup obtained by using our method compared to tested SAT solvers is also shown.

Instance	Satisfiable	Number of variables / number of clauses	MiniSAT (seconds)	zChaff (seconds)	HaifaSAT (second)
chnl10_11	unsat	220/1122	**34.30**	**7.54**	*> 600.00*
chnl10_12	unsat	240/1344	**101.81**	**9.11**	*> 600.00*
chnl10_13	unsat	260/1586	**200.30**	**11.47**	*> 600.00*
chnl11_12	unsat	264/1476	*> 600.00*	**33.49**	*> 600.00*
chnl11_13	unsat	286/1472	*> 600.00*	**187.08**	*> 600.00*
chnl11_20	unsat	440/4220	*> 600.00*	**329.57**	*> 600.00*
urq3_5	unsat	46/470	**95.04**	*> 600.00*	*> 600.00*
urq4_5	unsat	74/694	*> 600.00*	*> 600.00*	*> 600.00*
urq5_5	unsat	121/1210	*> 600.00*	*> 600.00*	*> 600.00*
urq6_5	unsat	180/1756	*> 600.00*	*> 600.00*	*> 600.00*
hole6	unsat	42/133	**0.01**	**0.01**	**0.01**
hole7	unsat	56/204	**0.09**	**0.04**	**0.02**
hole8	unsat	72/297	**0.49**	**0.23**	**0.94**
hole9	unsat	90/415	**3.64**	**1.46**	**478.16**
hole10	unsat	110/561	**39.24**	**7.53**	*> 600.00*
hole11	unsat	132/738	*> 600.00*	**32.36**	*> 600.00*
hole12	unsat	156/949	*> 600.00*	**372.18**	*> 600.00*
fpga10_11	unsat	220/1122	**44.77**	**12.58**	*> 600.00*
fpga10_12	unsat	240/1344	**119.26**	**33.82**	*> 600.00*
fpga10_13	unsat	260/1586	**362.24**	**76.15**	*> 600.00*
fpga10_15	unsat	300/2130	*> 600.00*	**274.84**	*> 600.00*
fpga10_20	unsat	400/3840	*> 600.00*	**546.00**	*> 600.00*
fpga11_12	unsat	264/1476	*> 600.00*	**55.70**	*> 600.00*
fpga11_13	unsat	286/1742	*> 600.00*	**237.54**	*> 600.00*
fpga11_14	unsat	308/2030	*> 600.00*	*> 600.00*	*> 600.00*
fpga11_15	unsat	330/2340	*> 600.00*	*> 600.00*	*> 600.00*
fpga11_20	unsat	440/4220	*> 600.00*	*> 600.00*	*> 600.00*

Table 4.1. EXPERIMENTAL COMPARISON OF SAT SOLVERS - PART I. Experimental comparison of three SAT solvers over the selected difficult benchmark SAT instances. We used the timeout of 10.0 minutes (600.00 seconds) for all the tests.

As it is evident from our experiments the proposed method brings significant improvement in terms of time necessary for the decision of the selected difficult benchmark problems (Pigeon holes, FPGA routing instances). The improvements are in the order of magnitude in comparison to all tested state-of-the-art SAT solvers. It seems that the improvement on selected benchmarks is exponential with respect to the best tested SAT solver. The conclusion is that there is still a space to improve SAT solvers. However, the

domain of the improvement is more likely in the difficult instances of SAT problems which are typically unsatisfiable.

Instance	Decided by preprocessing	Cliques (count x size)	Decision (seconds)	Speedup ratio w.r.t. MiniSAT	Speedup ratio w.r.t zChaff	Speedup ratio w.r.t HaifaSAT
chnl10_11	yes	20 x 11	**0.43**	**79.76**	**17.53**	*> 1395.34*
chnl10_12	yes	20 x 12	**0.60**	**169.68**	**8.51**	*> 1000.00*
chnl10_13	yes	20 x 13	**0.78**	**256.79**	**14.70**	*> 769.23*
chnl11_12	yes	22 x 12	**0.70**	*> 857.14*	**47.84**	*> 857.14*
chnl11_13	yes	22 x 13	**0.92**	*> 652.17*	**203.34**	*> 652.17*
chnl11_20	yes	22 x 20	**5.74**	*> 104.42*	**57.41**	*> 104.42*
urq3_5	no	47 x 2	**130.15**	**0.73**	*N/A*	*N/A*
urq4_5	no	73 x 2	*> 600.00*	*N/A*	*N/A*	*N/A*
urq5_5	no	120 x 2	*> 600.00*	*N/A*	*N/A*	*N/A*
urq6_5	no	179 x 2	*> 600.00*	*N/A*	*N/A*	*N/A*
hole6	yes	6 x 7	**0.01**	**1.0**	**1.0**	**1.0**
hole7	yes	7 x 8	**0.02**	**4.5**	**2.0**	**1.0**
hole8	yes	8 x 9	**0.04**	**12.25**	**5.75**	**23.5**
hole9	yes	9 x 10	**0.08**	**45.5**	**18.25**	**5977.00**
hole10	yes	10 x 11	**0.13**	**301.84**	**57.92**	*> 4615.38*
hole11	yes	11 x 12	**0.20**	*> 3000.00*	**161.8**	*> 3000.00*
hole12	yes	12 x 13	**0.30**	*> 2000.00*	**1240.6**	*> 2000.00*
fpga10_11	yes	20 x 11	**0.46**	**97.32**	**27.34**	*> 1304.34*
fpga10_12	yes	20 x 12	**0.64**	**186.34**	**52.84**	*> 937.50*
fpga10_13	yes	20 x 13	**0.84**	**431.23**	**90.65**	*> 714.28*
fpga10_15	yes	20 x 15	**1.39**	*> 431.65*	**197.72**	*> 431.65*
fpga10_20	yes	20 x 20	**4.72**	*> 127.11*	**115.67**	*> 127.11*
fpga11_12	yes	22 x 12	**0.76**	*> 789.47*	**73.28**	*> 789.47*
fpga11_13	yes	22 x 13	**1.01**	*> 594.05*	**235.18**	*> 594.05*
fpga11_14	yes	22 x 14	**1.30**	*> 461.53*	*> 461.53*	*> 461.53*
fpga11_15	yes	22 x 15	**1.67**	*> 359.28*	*> 359.28*	*> 359.28*
fpga11_20	yes	22 x 20	**5.96**	*> 100.67*	*> 100.67*	*> 100.67*

Table 4.2. EXPERIMENTAL COMPARISON OF SAT SOLVERS - PART II. Experimental comparison of three SAT solvers with the method using clique-consistency over the selected difficult benchmark SAT instances. Again timeout of 10.0 minutes (600.00 seconds) for all the tests was used.

It is also evident that the clique-consistency is not an universal method for difficult SAT instances. There is no improvement on instances where no cliques of reasonable size are found (randomized Urquhart instances). The interesting feature of the tested SAT instances is that they contain non-trivial cliques of the same size (there are also trivial cliques consisting of a single literal). This may be accounted to the symmetrical formulation of the problems.

In our further experiments we also performed the comparison with the RSAT solver (Pipatsrisawat and Darwiche, 2007). The results were very similar in the sense that the solver does not cope well with these problems. Unfortunately the solver is provided without the source code so we do not consider this test as a relevant one. Another SAT solver which worth consideration for our tests (achieved good results in the SAT Race competition (Sinz, 2006)) - Eureka (Nadel, 2007) - is not provided (neither source code nor executables are provided).

We also tested our approach on SAT instances where the preprocessing stage does not terminate by the answer that the given SAT instance cannot be satisfied. This is the situation when the problem is not decided by the preprocessing stage and a new equivalent SAT instance is produced and sent to the solver. In such situations our method does not provide competitive results. The resulting formula is typically solved faster by the SAT solver but the preprocessing stage takes too much time. The unaffordable time consumption in the preprocessing stage is caused by extensive propagation performed by the method by which huge numbers of conflicts are inferred. It seems that on these problems the proposed approach is too strong and represents an overhead only. The numbers of inferred conflicts is not proportional to the time saved in the search of the solution stage. However, this disadvantage may be overcome firstly by a better implementation of our technique (our current implementation is an experimental prototype and the quality of our code is uncompetitive with the quality of code of the tested SAT solvers) and secondly by making the propagation less extensive on problems with many conflicts (that is, not to infer all the conflicts).

The question may now be what to do with the method at the current stage of implementation when we have a new problem of unknown difficulty. That is, shall we use the method or the SAT solver of our choice directly? Technically we can answer this question as follows. We can run both the preprocessing method and the SAT solver in parallel. On a machine with more than one processor we obtain an exponential speedup (the method succeeds) or no improvement. On a machine with only one processor we may obtain an exponential speedup at the expense of constant slowdown. However, the ultimate goal of our implementing efforts is to answer this question automatically within the preprocessing phase.

4.3 Related Works

Our method for SAT problem reformulation was originally proposed for solving planning problems using planning graphs. Clique-consistency described in this chapter is an adaptation of projection consistency for the SAT domain.

The idea of exploiting structural information for solving problems is not new. There is a lot of works concerning this topic. Many of these works are dealing with methods for breaking symmetries (Aloul *et al.*, 2002; Benhamou and Sais, 1994; Crawford *et al.*, 1996). We share the goal with these methods, which is to reduce the search space. However, we differ in the way how we are doing this. We are rather trying to infer what would happen if the search over the problem proceeds in some way. And if that direction seems to be unpromising the corresponding part of the search space is skipped. Symmetry breaking methods are rather trying not to do the same work twice (or more times) by a clever transformation of the original problem.

Our work was much influenced by the paper of Aloul, Markov and Sakallah (2002). We are studying the same set of difficult SAT problems. Nevertheless, it seems that our method is simpler to implement and more effective on the set of selected testing problems.

Another original approach to solving SAT problems is to exploit integer programming (IP) techniques. An interesting combination of IP and SAT techniques is given in (Li *et al.*, 2004). The proposed IP approach is especially successful on difficult SAT problems.

Finally let us note that the detection of cliques in the structure of the problem is not new. A work dealing with a consistency based on cliques of inequalities was published by Sqalli and Freuder (1996). They use information about cliques to reach more global reasoning about the problem. Another work dealing with the similar ideas is (Frisch *et al.*, 2002) in which the authors use a graph structure of the problem to transform it into another formulation based on global constraints, which provide stronger propagation than the original formulation.

4.4 Summary and Conclusion

We proposed a method for preprocessing difficult (unsatisfiable) SAT instances based on the greedy clique decomposition of the transformed input CNF formula. Although the method is not universal it provides improvements in the order of magnitude compared to the state-of-the-art SAT solvers on tested SAT instances. Moreover, our method can be easily integrated into a SAT solver (new or existing) which may significantly improve its performance on difficult SAT instances.

For future we plan to further tune the method to be able to cope better with the problems having few edges in the graphs of conflicts (for example Urquhart instances). This may be done by some alternative consistency technique instead of singleton arc-consistency. We also plan to investigate the possibility to make the preprocessing iterative. That is to further preprocess the formula resulting from the previous preprocessing.

Another issue worth a deeper study is how the cliques of the clique decomposition should look like in order to our method can succeed. Our further experiments showed that better results can be obtained by using a clique decomposition where sizes of the individual cliques differ little (having several cliques of the similar size is better than having one large clique and several much smaller cliques).

We also plan to write an experimental SAT solver which would utilize the clique-consistency during search. This may be useful for early determining that a certain part of the search space does not contain a solution.

Finally, an interesting research direction is some kind of a combination of existing symmetry breaking methods and the proposed clique-consistency.

CHAPTER 5

CONCLUSIONS AND FUTURE WORKS

This thesis represents contribution to the areas of solving planning problems and solving Boolean satisfiability problems. The main results of this thesis are the improvements of the process of solving problem of supporting actions for a goal within the context of GraphPlan algorithm and special preprocessing method for solving Boolean satisfiability problems.

We gradually describe several variants of solving the problem of finding supporting actions for a goal within the GraphPlan algorithm. Solving of this problem is one of the weakest points of the whole algorithm. The hypothesis was that the improvement of this weak point may improve the algorithm as a whole. First, we proposed to model the problem as a constraint satisfaction problem and to maintain arc-consistency throughout the process of search for solution of the problem of finding supports. Several variants of propagation of arc-consistency were proposed. The GraphPlan algorithm enhanced by every variant of propagation of arc-consistency proved to be better than the original version in all the major measurable characteristics. The improvements were significant in terms of the overall solving time, in the number of constraint checks, and in the number of backtracks. These improvements were showed using experimental evaluation.

The promising results obtained by maintaining arc-consistency served as an inspiration for developing a specialized global consistency. This consistency exploits some structural properties of the problem of supports. Arc-consistency performs propagations only in a local neighborhood of currently explored part of the problem which seems to be weak. The idea was therefore to develop global consistency that would take into account the whole problem at once. These ideas put into reality are represented by the definition of projection consistency. This is a special type of consistency which exploits the structure of graphical representation of the problem of finding supporting actions. Namely, the consistency exploits the decomposition of the graph of the problem into several complete graphs - cliques. The knowledge of cliques decomposition allows us to use special counting arguments to rule out the actions from further considerations. It was theoretically shown that this ap-

proach can be stronger than local consistency such as arc-consistency. The performed experimental evaluation showed significant improvements compared to the best version that uses arc-consistency. The improvements were in the overall solving time as well as in other characteristics.

The final enhancement of the consistencies for solving the problem of finding supporting actions was the definition of a so called tractable class of the problem. It is in fact the class of problems defined using the proposed projection consistency that can be solved in polynomial time. This tractable class together with the heuristics that transform the general problem (not belonging to the class) to the problem belonging to the class forms the best performing algorithmic technique for solving the problem. The experimental evaluation showed that some planning problems can be solved even without backtracking using this enhancement within the GraphPlan algorithm. Compared to the previously developed techniques for solving the problem, the tractable class brings further improvements in comparison to the version which uses pure projection consistency.

The results achieved in solving Boolean satisfiability problems are represented by the special preprocessing technique which can be used to simplify the problems before they are passed to the general solving system. The success of exploiting the structural properties of the problem by projection consistency was an inspiration for finding similar structures in the Boolean satisfaction problems. The situation here was complicated by the fact that raw Boolean satisfaction problem lacks structures when it is interpreted graphically. Therefore several inference stages were proposed to make the structures visible and allow their detection and usage. The resulting consistency technique was called clique consistency since it again exploits cliques of the graphical interpretation of the problem. The clique consistency is in fact adaptation of projection consistency for Boolean satisfaction problems. The experimental evaluation showed that usage of clique consistency can outperform today's best Boolean satisfaction solvers (SAT solvers) on the selected set of difficult problems.

Despite the above described progress there is still a work for future. First, there is a possibility that projection consistency can be extended from a single layer of the planning graph on the whole planning graph during the search. Some preliminary experiments show a significant reduction of the number of backtracks but the overall solving time did not improve or yet worsened. The reasons were definitely in the implementation since the effective implementation of this algorithm requires non-trivial efforts. These preliminary tests were done with an implementation that extensively performs inefficient re-computations.

The next interesting research relates to the definition of projection consistency. We found that the consistency can be made stronger by more careful calculation of counting arguments. To be more precise, it is possible to compute counting argument as a maximum network flow which is more accurate than calculating ordinary sum (sum relaxes the problem more). Computing counting arguments as maximum network flows still relaxes the problem but not that much as the original version. The open question that remains is what an improvement can be gained by using this more accurate computation.

BIBLIOGRAPHY

Harold **Abelson**, Gerald Jay **Sussman**, and Julie **Sussman (1985)**. *Structure and Interpretation of Computer Programs*. McGraw-Hill Book Company.

Ravindra K. **Ahuja**, Thomas L. **Magnanti**, and James B. **Orlin (1993)**. *Network Flows: Theory, Algorithms and Applications.* Prentice Hall.

Mitch **Ai-Chang**, John **Bresina**, Leonard **Charest**, Jennifer **Hsu**, Ari K. **Jónsson**, Bob **Kanefsky**, Pierre **Maldague**, Paul **Morris**, Kanna **Rajan**, and Jeffrey **Yglesias (2004)**. *MAPGEN: Mixed-Initiative Planning and Scheduling for the Mars Exploration Rover Mission.* IEEE Intelligent Systems, Volume 19(1), 8-12, IEEE Press.

James F. **Allen (1983)**. *Maintaining Knowledge about Temporal Intervals*. Communications of the ACM (CACM), Volume 26 (11), 832-843, ACM Press.

James F. **Allen** and Johannes A. G. M. **Koomen (1983)**. *Planning Using a Temporal World Model*. In Proceedings of the 8th International Joint Conference on Artificial Intelligence. (IJCAI 1983), Karlsruhe, West Germany, 741-747, William Kaufmann.

James F. **Allen**, James A. **Hendler**, and Austin **Tate** (editors). **(1990)**. *Readings in Planning.* Morgan Kaufmann Publishers.

James F. **Allen (1991)**. *Planning as Temporal Reasoning*. In Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR 1991), Cambridge, MA, USA, 3-14, Morgan Kaufmann Publishers.

Fadi A. **Aloul**, Arathi **Ramani**, Igor L. **Markov**, and Karem A. **Sakallah (2002)**. *Solving difficult SAT instances in the presence of symmetry*. Proceedings of the 39th Design Automation Conference (DAC 2002), New Orleans, LA, USA, 731-736, ACM Press.

Fadi A. **Aloul (2007)**. *Fadi Aloul's Home Page - SAT Benchmarks*. Personal Web Page. http://www.eecs.umich.edu/~faloul/benchmarks.html, University of Michigan, MI, USA, (March 2007).

Fahiem **Bacchus** and Michael **Ady (2001)**. *Planning with Resources and Concurrency: A Forward Chaining Approach*. In Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, USA, 417-424, Morgan Kaufmann Publishers.

Christer **Bäckström** and Bernhard **Nebel (1992)**. O*n the Computational Complexity of Planning and Story Understanding*. In Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 1992), Vienna, Austria, 349-353, John Wiley and Sons.

Marco **Baioletti**, Stefano **Marcugini**, and Alfredo **Milani (1998)**. *An Extension of SAT-PLAN for Planning with Constraints*. Proceedings of Artificial Intelligence: Methodology, Systems, and Applications, 8th International Conference, (AIMSA 1998), Sozopol, Bulgaria, 39-49, LNCS 1480, Springer Verlag.

Andrew B. **Baker (1995)**. *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. Doctoral Thesis, University of Oregon, Eugene, OR, USA.

Philippe **Baptiste**, Claude **Le Pape**, and Wim **Nuijten (2001)**. *Constraint-Based Scheduling*. Kluwer Academic Publishers.

Roman **Barták** and Radek **Erben (2004)**. *A New Algorithm for Singleton Arc Consistency*. Proceedings of the 17th International Florida Artificial Intelligence Research Society Conference (FLAIRS 2004), Miami Beach, FL, USA, AAAI Press.

Belaid **Benhamou** and Lakhdar **Sais (1994)**. *Tractability through Symmetries in Propositional Calculus*. Journal of Automated Reasoning (JAR), Volume 12 (1), 89-102, Springer Verlag.

Christian **Bessière (1992)**. *Arc-Consistency for Non-Binary Dynamic CSPs.* Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 1992), Vienna, Austria, 23-27, John Wiley and Sons.

Christian **Bessière** and Marie-Odile **Cordier (1993)**. *Arc-Consistency and Arc-Consistency Again.* Proceedings of the 11th National Conference on Artificial Intelligence (AAAI 1993), Washington, DC, USA, 108-113, AAAI Press/MIT Press.

Christian **Bessière** and Romuald **Debruyne (2005)**. *Optimal and Suboptimal Singleton Arc Consistency Algorithms*. Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005), Edinburgh, Scotland, United Kingdom, 54-59, Professional Book Center.

Christian **Bessière** and Jean-Charles **Régin (1996)**. *MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems.* Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming (CP 1996), Cambridge, MA, USA, 61-75, LNCS 1118, Springer Verlag.

Christian **Bessière** and Jean-Charles **Régin (2001)**. *Refining the Basic Constraint Propagation Algorithm.* Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, USA, 309-315, Morgan Kaufmann Publishers.

Avrim L. **Blum** and Merrick L. **Furst (1997)**. *Fast planning through planning graph analysis.* Artificial Intelligence, Volume 90 (1-2), 281-300, AAAI Press.

Avrim L. **Blum** and John **Langford (2000)**. *Probabilistic Planning in the Graphplan Framework*. Recent Advances in AI Planning, Proceedings of the 5th European Conference on Planning (ECP 1999), Durham, UK, 319-332, LNCS 1809, Springer Verlag.

Boeing Corporation **(2003a)**. *UCAV-N Naval Unmanned Combat Air Vehicle.* Commercial web page. http://www.boeing.com/defense-space/military/unmanned/ucav-n.html, Boeing Co., USA, (May 2007).

Boeing Corporation **(2003b)**. *Unmanned Combat Air Vehicle (X-45).* Commercial web page. http://www.boeing.com/phantom/ucav.html, Boeing Co., USA, (May 2007).

Blai **Bonet** and Hector **Geffner (2001a)**. *Planning as heuristic search*. Artificial Intelligence, Volume 129 (1-2), 5-33, Elsevier Science Publishers.

Blai **Bonet** and Hector **Geffner (2001b)**. *Heuristic Search Planner 2.0*. AI Magazine, Volume 22 (3), 77-80, AAAI Press.

Tom **Bylander (1994)**. *The Computational Complexity of Propositional STRIPS Planning.* Artificial Intelligence, Volume 69 (1-2), 165-204, Elsevier Science Publishers.

CHOCO (2007). *CHOCO Solver*. Project web page, http://choco-solver.net, (July 2007).

Stephen A. **Cook (1971)**. *The Complexity of Theorem-Proving Procedures.* Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971), Shaker Heights, OH, USA, 151-158, ACM Press.

James M. **Crawford**, Matthew L. **Ginsberg**, Eugene M. **Luks**, and Amitabha **Roy (1996)**. *Symmetry-Breaking Predicates for Search Problems*. Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR 1996), Cambridge, MA, USA, 148-159, Morgan Kaufmann.

Ken **Currie** and Austin **Tate (1991)**. *O-Plan: The open Planning Architecture*. Artificial Intelligence, Volume 52 (1), 49-86, Elsevier Science Publishers.

The Defense Advanced Research Projects Agency - **DARPA (2007a)**. *DARPA Grand Challenge*. Competition web page, http://www.darpa.mil/GrandChallenge/index.asp, DARPA, USA, (June 2007).

The Defense Advanced Research Projects Agency - **DARPA (2007b)**. *DARPA Urban Challenge*. Competition web page, http://www.darpa.mil/GrandChallenge/overview.asp, DARPA, USA, (June 2007).

Martin **Davis** and Hilary **Putnam (1960)**. *A Computing Procedure for Quantification Theory.* Journal of the ACM, Volume 7 (3), 201-215, ACM Press.

Rina **Dechter**, Itay **Meiri**, and Judea **Pearl (1989)**. *Temporal Constraint Networks*. In Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR 1989), Toronto, Canada, 83-93, Morgan Kaufmann Publishers.

Rina **Dechter (2003)**. *Constraint Processing*. Morgan Kaufmann Publishers.

Minh Binh **Do** and Subbarao **Kambhampati (2001)**. *Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP*. Artificial Intelligence, Volume 132 (2), 151-182, Elsevier Science Publishers.

William F. **Dowling** and Jean H. **Gallier (1984)**. *Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae*. Journal of Logic Programming, Volume 1 (3), 267-284, Elsevier Science Publishers.

Stefan **Edelkamp**, Jörg **Hoffmann**, Michael **Littman**, Hakan **Younes**, Fahiem **Bacchus**, Drew **McDermott**, Maria **Fox**, Derek **Long**, Jussi **Rintanen**, David **Smith**, Sylvie **Thiebaux**, and Daniel **Weld (2004)**. *The 2004 International Planning Competition*. Event in the context of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004), Whistler, British Columbia, Canada, http://andorfer.cs.uni-dortmund.de/~edelkamp/ipc-4/, University of Dortmund, Germany, (June 2007).

Niklas **Eén** and Niklas **Sörensson (2005)**. *MiniSat — A SAT Solver with Conflict-Clause Minimization*. Poster in 8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005), Scotland, 2005.

Niklas **Eén** and Niklas **Sörensson (2007)**. *The MiniSat Page*. Research web page, http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/Main.html, Chalmers University, Sweden, (March 2007).

Amin **El-Kholy** and Barry **Richards (1996)**. *Temporal and Resource Reasoning in Planning: the parcPLAN approach*. In Proceedings of the 12th European Conference on Artificial Intelligence (ECAI 1996), Budapest, Hungary, 614-618, John Wiley and Sons.

Kutluhan **Erol**, James A. **Hendler**, and Dana S. **Nau (1994a)**. *UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning*. In Proceedings of the 2nd International Conference on Artificial Intelligence Planning Systems (AIPS 1994), Chicago, IL, USA, 249-254, AAAI Press.

Kutluhan **Erol**, James A. **Hendler**, Dana S. **Nau (1994b)**. *HTN Planning: Complexity and Expressivity*. In Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994), Seattle, WA, USA, Volume 2, 1123-1128, AAAI Press.

Kutluhan **Erol**, Dana S. **Nau**, V. S. **Subrahmanian (1995)**. *Complexity, Decidability and Undecidability Results for Domain-Independent Planning*. Artificial Intelligence, Volume 76 (1-2), 75-88, Elsevier Science Publishers.

Kutluhan **Erol**, James A. **Hendler**, and Dana S. **Nau (1996)**. *Complexity Results for HTN Planning.* Annals of Mathematics and Artificial Intelligence, Volume 18 (1), 69-93, Springer Verlag.

Richard **Fikes** and Nils J. **Nilsson (1971)**. *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving.* Artificial Intelligence, Volume 2 (3/4), 189-208, Elsevier Science Publishers.

Maria **Fox** and Derek **Long (2003)**. *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains.* Journal of Artificial Intelligence Research (JAIR), Volume 20, 61-124, AAAI Press.

Jeremy **Frank**, Ari K. **Jonsson**, Robert **Morris**, and David E. **Smith (2001)**. *Planning and scheduling for fleets of earth observing satellites*. In Proceedings of the 6th International Symposium on AI, Robotics and Automation for Space (i-SAIRAS 2001), http://robotics.estec.esa.int/i-SAIRAS, (May 2007).

Jeremy **Frank** and Ari K. **Jónsson (2003)**. *Constraint-Based Attribute and Interval Planning*. Constraints, Volume 8 (4), 339-364, Springer Verlag.

Alan M. **Frisch**, Ian **Miguel**, and Toby **Walsh** (2002). *CGRASS: A System for Transforming Constraint Satisfaction Problems*. Recent Advances in Constraints, Proceedings of the Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2002), Cork, Ireland, 15-30, LNCS 2627, Springer Verlag.

Zhaohui **Fu**, Yogesh **Marhajan**, and Sharad **Malik (2007)**. *zChaff*. Research Web Page. http://www.princeton.edu/ ~chaff/zchaff.html, Princeton University, USA, (March 2007).

Gecode (2007). *generic constraint development environment*. Project web page, http://www.gecode.org/, (July 2007).

Alfonso **Gerevini**, Yannis **Dimopoulos**, Patrik **Haslum**, and Alessandro **Saetti (2006)**. *5th International Planning Competition*. Event in the context of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006), Cumbria, UK, http://ipc5.ing.unibs.it, University of Brescia, Italy, (May 2007).

Alfonso **Gerevini** and Ivan **Serina (2002)**. *LPG: a Planner based on Local Search for Planning Graphs*. In Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02), Toulouse, France, 13-22, AAAI Press, 2002.

Alfonso **Gerevini** and Ivan **Serina (2007)**. *Homepage of LPG*. Research web page, http://zeus.ing.unibs.it/lpg/, University of Brescia, Italy, (April 2007).

Roman **Gershman** and Ofer **Strichman (2005)**. *HaifaSat: A New Robust SAT Solver*. Hardware and Software Verification and Testing, First International Haifa Verification Conference (Haifa Verification Conference 2005), Haifa, Israel, 76-89, LNCS 3875, Springer Verlag.

Roman **Gershman** and Ofer **Strichman (2007)**. *HaifaSat – a new robust SAT solver*. Research Web Page. http://www.cs.technion.ac.il/~gershman/HaifaSat.htm, Technion Haifa, Israel, (March 2007).

Malik **Ghallab**, Dana S. **Nau**, and Paolo **Traverso (2004)**. *Automated Planning: theory and practice.* Morgan Kaufmann Publishers.

GNU Project (2008). *GCC, the GNU Compiler Collection.* Project Web Page. http://gcc.gnu.org/, (March 2008).

Martin C. **Golumbic (1980)**. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press.

William D. **Harvey (1995)**. *Nonsystematic Backtracking Search*. Doctoral Thesis, University of Oregon, Eugene, OR, USA.

Malte **Helmert (2003)**. *Complexity results for standard benchmark domains in planning*. Artificial Intelligence, Volume 143 (2), 219-262, Elsevier Science Publishers.

Malte **Helmert (2006)**. *New Complexity Results for Classical Planning Benchmarks.* In Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006), Cumbria, UK, 52-61, AAAI Press.

Chih-Wei **Hsu**, Benjamin W. **Wah**, Ruoyun **Huang**, and Yixin **Chen (2006)**. *Handling Soft Constraints and Preferences in SGPlan*. In Proceedings of the ICAPS Workshop on Preferences and Soft Constraints in Planning, event in the context of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006), Cumbria, UK, 2006.

Chih-Wei **Hsu**, Benjamin W. **Wah**, Ruoyun **Huang**, and Yixin **Chen (2007)**. *SGPlan 5: Subgoal Partitioning and Resolution in Planning*. Research web page. http://manip.crhc.uiuc.edu/programs/SGPlan/index.html, University of Illinois, USA, (April 2007).

ILOG SA **(2007)**. *ILOG Solver*. Commercial web page, http://www.ilog.com, ILOG SA, France, (July 2007).

Ari K. **Jónsson**, Paul H. **Morris**, Nicola **Muscettola**, Kanna **Rajan**, and Benjamin D. **Smith (2000)**. *Planning in Interplanetary Space: Theory and Practice*. In Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS 2000), Breckenridge, CO, USA, 177-186, AAAI Press.

Narendra **Jussien**, Romuald **Debruyne**, and Patrice **Boizumault (2000)**. *Maintaining Arc-Consistency within Dynamic Backtracking*. Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000), Singapore, 249-261, LNCS 1894, Springer Verlag.

Leslie Pack **Kaelbling**, Michael L. **Littman**, Anthony R. **Cassandra (1995)**. *Partially Observable Markov Decision Processes for Artificial Intelligence*. Advances in Artificial Intelligence, Proceedings of the 19th Annual German Conference on Artificial Intelligence (KI 1995), Bielefeld, Germany, 1-17, LNCS 981, Springer Verlag.

Leslie Pack **Kaelbling**, Michael L. **Littman**, Anthony R. **Cassandra (1998)**. *Planning and Acting in Partially Observable Stochastic Domains*. Artificial Intelligence, Volume 101 (1-2), 99-134, Elsevier Science Publishers.

Subbarao **Kambhampati**, Eric **Parker**, and Eric **Lambrecht (1997)**. *Understanding and Extending Graphplan*. Proceedings of the 4th European Conference on Planning (ECP 1997), Toulouse, France, 260-272, LNCS 1348, Springer Verlag.

Subarao **Kambhampati (2000)**. *Planning Graph as a (Dynamic) CSP: Exploiting EBL, DDB and other CSP Search Techniques in GraphPlan*. Journal of Artificial Intelligence Research 12 (JAIR 12), 1-34, AAAI Press.

Henry A. **Kautz**, David A. **McAllester**, and Bart **Selman (1996)**. *Encoding Plans in Propositional Logic*. Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR 1996), Cambridge, MA, USA, 374-384, Morgan Kaufmann Publishers.

Henry A. **Kautz** and Bart **Selman (1992)**. *Planning as Satisfiability*. Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 1992), Vienna, Austria, 359-363, John Wiley and Sons.

Henry A. **Kautz** and Bart **Selman (1999)**. *Unifying SAT-based and Graph-based Planning*. In Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999), Stockholm, Sweden, 318-325, Morgan Kaufmann Publishers.

Henry A. **Kautz**, Bart **Selman**, and Jörg **Hoffmann (2006)**. *SATPlan: Planning as Satisfiability*. Abstracts of the 5th International Planning Competition, event in the context of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006), Cumbria, UK, http://ipc5.ing.unibs.it, University of Brescia, Italy, (May 2007).

Henry A. **Kautz**, Bart **Selman**, and Jörg **Hoffmann (2007)**. *SATPLAN*. Research web page. http://www.cs.rochester.edu/u/kautz/satplan/index.htm, University of Rochester, NY, USA, (April 2007).

Jana **Koehler (2007)**. *Homepage of IPP*. Research web page, http://www.informatik.uni-freiburg.de/~koehler/ipp.html, University of Freiburg, Germany, (April 2007).

Jana **Koehler**, Bernhard **Nebel**, Jörg **Hoffmann**, and Yannis **Dimopoulos (1997)**. *Extending Planning Graphs to an ADL Subset*. In Proceedings of the 4th European Conference on Planning (ECP-97), Toulouse, France, 273-285, LNAI 1348, Springer-Verlag.

Satish **Kumar** Thittamaranahalli **(2005)**. *Contributions to Algorithmic Techniques in Automated Reasoning about Physical Systems.* Doctoral Dissertation, Stanford University, CA, USA.

Philippe **Laborie** and Malik **Ghallab (1995)**. *Planning with Sharable Resource Constraints*. In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 1995), Montréal, Québec, Canada, 1643-1651, Morgan Kaufmann Publishers.

Daniel **Le Berre** and Laurent **Simon (2005)**. *SAT Competition 2005*. Competition Web Page, http://www.satcompetition.org/2005/, Scotland, (March 2007).

Ruiming **Li**, Dian **Zhou**, and Donglei **Du (2004)**. *Satisfiability and integer programming as complementary tools*. Proceedings of the 2004 Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair 2004 (ASP-DAC 2004), 879-882, Japan, IEEE Press.

Vassilis **Liatsos** and Barry **Richards (1999)**. *Scaleability in Planning*. Recent Advances in AI Planning, 5th European Conference on Planning (ECP 1999), Durham, UK, 49-61, LNCS 1809, Springer Verlag.

Iain **Little**, Douglas **Aberdeen**, and Sylvie **Thiébaux (2005)**. *Prottle: A Probabilistic Temporal Planner*. In Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005) and the 17th Innovative Applications of Artificial Intelligence Conference (IAAI 2005), Pittsburgh, PA, USA, 1181-1186, AAAI Press / The MIT Press.

Iain **Little** and Sylvie **Thiébaux (2006)**. *Concurrent Probabilistic Planning in the Graphplan Framework*. In Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006), Cumbria, UK, 263-272, AAAI Press.

Derek **Long** and Maria **Fox (1999)**. *Efficient Implementation of the Plan Graph in STAN*. Journal of Artificial Intelligence Research, Volume 10, 87-115, AAAI Press.

Derek **Long** and Maria **Fox (2003)**. *Exploiting a Graphplan Framework in Temporal Planning*. Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS 2003), Trento, Italy, 52-61, AAAI Press.

Derek **Long**, Maria **Fox**, David E. **Smith**, Drew **McDermott**, Fahiem **Bacchus**, and Hector **Geffner (2002)**. *The 2002 International Planning Competition*. Event in the context of the 6th International Conference on AI Planning & Scheduling (AIPS 2002), Toulouse, France, 2002, http://planning.cis.strath.ac.uk/competition/, University of Strathclyde, UK, (June 2007).

Adriana **Lopez** and Fahiem **Bacchus (2003)**. *Generalizing GraphPlan by Formulating Planning as a CSP*. Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003), Acapulco, Mexico, 954-960, Morgan Kaufmann Publishers.

Alan K. **Mackworth (1977)**. *Consistency in Networks of Relations*. Artificial Intelligence, , Volume 8 (1), 99-118, Elsevier Science Publishers.

Mandriva (2008). *Mandriva 10th Year of Innovation.* Commercial Web page. http://www.mandriva.com/, (March, 2008).

Drew **McDermott (1998)**. *PDDL: the Planning Domain Definition Language.* Technical Report. Yale Center for Computational Vision and Control, Yale University, CT, USA, 1998.

Roger **Mohr** and Thomas C. **Henderson (1986)**. *Arc and Path Consistency Revisited.* Artificial Intelligence, Volume 28 (2), 225-233, Elsevier Science Publishers.

Matthew W. **Moskewicz**, Conor F. **Madigan**, Ying **Zhao**, Lintao **Zhang**, and Sharad **Malik (2001)**. *Chaff: Engineering an Efficient SAT Solver*. Proceedings of the 38th Design Automation Conference (DAC-2001), Las Vegas, NV, USA, 530-535, ACM Press.

Héctor **Muñoz-Avila**, David W. Aha, Dana S. **Nau**, Rosina **Weber**, Len **Breslow**, and Fusun **Yaman (2001)**. *SiN: Integrating Case-based Reasoning with Task Decomposition*. In Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, USA, 999-1004, Morgan Kaufmann Publishers.

Nicola **Muscettola**, P. Pandurang **Nayak**, Barney **Pell,** and Brian **Williams (1998)**. *Remote Agent: To Boldly Go Where No AI System Has Gone Before*. Artificial Intelligence, Volume 103 (1-2), 5-48, Elsevier Science Publishers.

Alexander **Nadel (2007)**. *Alexander Nadel's Page*. Research Web Page. http://www.cs.tau.ac.il/~ale1/, Tel Aviv University, Israel, (March 2007).

Gi-Joon **Nam**, Karem A. **Sakallah**, and Rob A. **Rutenbar** (2002). *A new FPGA detailed routing approach via search-based Boolean satisfiability*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Volume 21 (6), 674-684, IEEE Press.

Alexander **Nareyek**, Eugene C. **Freuder**, Robert **Fourer**, Enrico **Giunchiglia**, Robert P. **Goldman**, Henry A. **Kautz**, Jussi **Rintanen**, and Austin **Tate (2005)**. *Constraints and AI Planning*. IEEE Intelligent Systems, Volume 20 (2), 62-72, 2005, IEEE Press.

Dana S. **Nau**, Satyandra K. **Gupta**, William C. **Regli (1995)**. *AI Planning Versus Manufacturing-Operation Planning: A Case Study*. In Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 1995), Montréal, Québec, Canada, 1670-1676, Morgan Kaufmann Publishers.

Dana S. **Nau**, Héctor **Muñoz-Avila**, Yue **Cao**, Amnon **Lotem**, and Steven **Mitchell (2001)**. *Total-Order Planning with Partially Ordered Subtasks*. In Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, USA, 425-430, Morgan Kaufmann Publishers.

P. Pandurang **Nayak**, Douglas E. **Bernard**, Gregory **Dorais**, Edward B. **Gamble Jr.**, Bob **Kanefsky**, James **Kurien**, William **Millar**, Nicola **Muscettola**, Kanna **Rajan**, Nicolas **Rouquette**, Benjamin D. **Smith**, William **Taylor**, and Yu-wen **Tung (1999)**. V*alidating the Deep Space 1 Remote Agent Experiment*. In Proceedings of the 5th International Symposium on Artificial Intelligence, Robotics and Automation in Space (iSAIRAS 1999), http://robotics.estec.esa.int/i-SAIRAS, (June 2007).

XuanLong **Nguyen**, Subbarao **Kambhampati**, and Romeo Sanchez **Nigenda (2002)**. *Planning Graph as the Basis for Deriving Heuristics for Plan Synthesis by State Space and CSP Search*. Artificial Intelligence, Volume 135 (1-2), 73-123, Elsevier Science Publishers.

Christos H. **Papadimitriou (1994)**. *Computational Complexity*. Addison Wesley.

Knot **Pipatsrisawat** and Adnan **Darwiche (2007)**. *RSat - ...veRSATile...* Research Web Page. http://reasoning.cs.ucla.edu/rsat/, University of California Los Angeles, CA, USA, (March 2007).

Edwin P. D. **Pednault** (1989). *ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus*. In Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR 1989), Toronto, Canada, 324-332, Morgan Kaufmann Publishers.

Jean-Charles **Régin (1994)**. *A Filtering Algorithm for Constraints of Difference in CSPs.* Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994), Seattle, WA, USA, 362-367, AAAI Press.

Stuart **Russell** and Peter **Norvig (2003)**. *Artificial Intelligence: A Modern Approach (second edition)*. Prentice Hall.

Lawrence **Ryan (2007)**. *the siege sat solver*. Research web page, http://www.cs.sfu.ca/~cl/software/siege/, Computational Logic Laboratory, Simon Fraser University, British Columbia, Canada (June 2007).

Christian **Schulte (2002)**. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*. Springer Verlag.

SICStus (2007). *SICStus Prolog - Leading Prolog Technology*. Commercial web page, http://www.sics.se/isl/sicstuswww/site/index.html, Swedish Institute of Computer Science, Sweden, (July 2007).

Carsten **Sinz (2006)**. *SAT-Race 2006*. Competition Web Page, http://fmv.jku.at/sat-race-2006/, USA, (March 2007).

David E. **Smith** and Daniel S. **Weld (1999)**. *Temporal Planning with Mutual Exclusion Reasoning.* In Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999), Stockholm, Sweden, 326-337, Morgan Kaufmann Publishers.

Mohammed H. **Sqalli** and Eugene C. **Freuder (1996)**. *Inference-Based Constraint Satisfaction Supports Explanation*. Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference, (AAAI/IAAI 1996), Portland, OR, USA, 318-325, AAAI Press / The MIT Press.

Bjarne **Stroustrup (1986)**. *The C++ Programming Language.* Addison-Wesley.

Pavel **Surynek (2003)**. *Solving Dynamic Constraint Satisfaction Problems*. Diploma thesis, 100 pages, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic.

Pavel **Surynek (2005)**. *Dynamic Constraint Satisfaction Problems*. Thesis to fulfill requirements of the degree of Doctor of Natural Sciences (RNDr.), 180 pages, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic.

Pavel **Surynek (2006)**. *Constraint Based Reasoning over Mutex Relations in GraphPlan Algorithm.* In Proceedings of the 11th Annual ERCIM Workshop on Constraint Solving and Constraint Programming (CSCLP 2006), Caparica, Portugal, 231-245, University of Lisbon.

Pavel **Surynek (2007a)**. *Constraint Based Reasoning over Mutex Relations in Planning Graphs during Search.* Technical Report, ITI Series, 2007-329, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic.

Pavel **Surynek (2007b)**. *Maintaining Arc-consistency over Mutex Relations in Planning Graphs during Search.* Technical Report, ITI Series, 2007-328, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic.

Pavel **Surynek (2007c)**. *Projection Global Consistency: An Application in AI Planning*. Proceedings of the 12th Annual ERCIM Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2007), Rocquencourt, France, 61-75, INRIA, 2007.

Pavel **Surynek (2007d)**. *Projection Global Consistency: An Application in AI Planning.* Technical Report, ITI Series, 2007-333, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic.

Pavel **Surynek (2007e)**. *Tractable Classes of a Problem of Finding Supporting Actions for a Goal in AI Planning.* Technical Report, ITI Series, 2007-338, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic.

Pavel **Surynek (2007f)**. *Solving Difficult SAT Instances Using Greedy Clique Decomposition*. Proceedings of the 7th International Symposium Abstraction, Reformulation, and Approximation (SARA 2007), Whistler, Canada. 359-374, Lecture Notes in Computer Science 4612, Springer Verlag.

Pavel **Surynek (2007g)**. *Solving Difficult SAT Instances Using Greedy Clique Decomposition*. Technical Report, ITI Series, 2007-340, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic.

Pavel **Surynek (2008a)**. *A Global Filtration for Satisfying Goals in Mutual Exclusion Networks*. Recent Advances in Constraints 2007 (RAC-2007), Post-proceedings of CSCLP 2007 Workshop, Lecture Notes in Artificial Intelligence, 5129, Springer Verlag.

Pavel **Surynek (2008b)**. *Tractable Class of a Problem of Goal Satisfaction in Mutual Exclusion Network.* In Proceedings of the 21st International Florida Artificial Intelligence Research Society Conference (FLAIRS-2008), Miami, FL, USA, 561-566, AAAI Press.

Pavel **Surynek** and Roman **Barták (2005)**. *Encoding HTN Planning as a Dynamic CSP*. In Proceedings Principles and Practice of Constraint Programming, 11th International Conference, (CP 2005), Sitges (Barcelona), Spain, 868, LNCS 3909, Springer Verlag.

Pavel **Surynek** and Roman **Barták (2007a)**. *Maintaining Arc-consistency over Mutex Relations in Planning Graphs during Search.* In Proceedings of the 20th International Florida Artificial Intelligence Research Society Conference (FLAIRS-2007), Key West, FL, USA, 134-139, AAAI Press.

Pavel **Surynek** and Roman **Barták (2007b)**. *Tractable Class of a Problem of Finding Supports.* Proceedings of the Doctoral Programme of the 13th International Conference of Principles and Practice of Constraint Programming, Providence, RI, 169-174, USA.

Pavel **Surynek**, Lukáš **Chrpa**, and Jiří **Vyskočil (2007a)**. *Solving Difficult Problems by Viewing Them as Structured Dense Graphs.* Proceedings of the 3rd Indian International Conference on Artificial Intelligence (IICAI 2007), Pune, India.

Pavel **Surynek**, Lukáš **Chrpa**, and Jiří **Vyskočil (2007b)**. *Solving Difficult Problems by Viewing Them as Structured Dense Graphs.* Technical Report, ITI Series, 2007-350, Faculty of Mathematics and Physics, Charles University, Prague, Czech Republic.

Sebastian **Thrun**, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, Kenny Lau, Celia Oakley, Mark Palatucci, Vaughan Pratt, Pascai Stang, Sven Strohband, Cedric Dupont, Lars-Erik Jendrossek, Christian Koelen, Charles Markey, Carlo Rummel, Joe van Niekerk, Eric Jensen, Philippe Alessandrini, Gary Bradski, Bob Davies, Scott Ettinger, Adrian Kaehler, Ara Nefian, and Pamela Mahoney **(2006)**. *Stanley, the robot that won the DARPA Grand Challenge.* Journal of Field Robotics,Volume 23, John Wiley and Sons.

Alasdair **Urquhart (1987)**. *Hard examples for resolution*. Journal of the ACM, Volume 34, 209-219, ACM Press.

Peter **van Beek** and Xinguang **Chen (1999)**. *CPlan: A Constraint Programming Approach to Planning*. In Proceedings of the 16th National Conference on Artificial Intelligence and 11th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI 1999), Orlando, Florida, USA, 585-590, AAAI Press / The MIT Press.

Miroslav N. **Velev** and Randal E. **Bryant (2003)**. *Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors.* Journal of Symbolic Computation (JSC), Volume 35 (2), 73-106, Elsevier.

Vincent **Vidal** and Hector **Geffner (2004)**. *Branching and Pruning: An Optimal Temporal POCL Planner-based on Constraint Programming.* In Proceedings of the AAAI Workshop on Integrating Planning Into Scheduling, event in the context of the AAAI 2004 conference, USA.

Vincent **Vidal** and Hector **Geffner (2007)**. *CPT Description.* Research web page, http://www.cril.univ-artois.fr/~vidal/cpt.html, Université D'Artois, France, (April 2007).

Marc B. **Vilain** and Henry A. **Kautz (1986)**. *Constraint Propagation Algorithms for Temporal Reasoning*. In Proceedings of the 5th National Conference on Artificial Intelligence (AAAI 1986), Philadelphia, PA, USA, 377-382, Morgan Kaufmann Publishers.

David L. **Waltz (1975)**. *Understanding line drawings of scenes with shadows.* The Psychology of Computer Vision, P. Winston (editor), 19–91, McGraw-Hill, New York.

Hantao **Zhang** and Mark E. **Stickel (1996)**. *An efficient algorithm for unit propagation*. Proceedings of the 4th International Symposium on Artificial Intelligence and Mathematics (MATH 1996), Fort Lauderdale, FL, USA, 1996.

Yuanlin **Zhang** and Roland H. C. **Yap (2001)**. *Making AC-3 an Optimal Algorithm.* Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA, USA, 316-321, Morgan Kaufmann Publishers.

Xing **Zhao**, Yixin **Chen**, and Weixiong **Zhang (2006)**. *MaxPlan: Optimal Planning by Decomposed Satisfiability and Backward Reduction*. Abstracts of the 5th International Planning Competition, Event in the context of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006), Cumbria, UK, http://ipc5.ing.unibs.it, University of Brescia, Italy, (May 2007).

Xing **Zhao**, Yixin **Chen**, Weixiong **Zhang**, and Ruoyun **Huang (2007)**. *MaxPlan.* Research web page, http://www.cse.wustl.edu/~chen/maxplan/, Washington University in St. Louis, MO, USA, (April 2007).

Terry **Zimmerman** and Subbarao **Kambhampati (2005)**. *Using Memory to Transform Search on the Planning Graph*. Journal of Artificial Intelligence Research (JAIR), Volume 23, 533-585, AAAI Press.

APPENDIX A

DIFFICULT PLANNING PROBLEMS

This appendix describes the set of difficult planning problems used for competitive experimentation in chapter 3. We used three classes of planning problems that encodes an unsolvable Dirichlet's box principle in some form. At the same time we show encoding planning problems using *Planning Domain Description Language* (*PDDL*). PDDL became a standard language for expressing planning problems (McDermott, 1998).

The PDDL language in its basic form uses the classical representation of planning problems (with some technical extensions). The problem is stated by the description of an initial state and a goal and by the description of a set of allowed planning operators. The description of the input planning problem consists of two files. One of these files contains description of a *planning domain*. In the terminology of PDDL the planning domain is the description of the language (the list of constants, the list of predicates etc.) and a set of allowed planning operators. The second file contains the description of an initial state and a goal. The splitting of the input into these two files is done because an initial state and a goal represents a very variable part of the input while for a certain planning environment the domain typically remains the same.

The syntax of the *PDDL* language is based on the *LISP* language (Abelson *et al.*, 1985) which is a representative of the languages based on the functional paradigm. The *PDDL* language itself became a standard for the inputs of many planning systems. It is also the standard for the input of planning problems of the *IPC* - International Planning Competition (Gerevini *et al.*, 2006).

There are also extensions of the language for handling time and resources (Fox and Long, 2003). The newer versions of the language also provide tools for expressing quality of the solution plans which does not have to be the least number of time-steps. It is possible for instance to write a planning problem where the objective is to spare certain type of resource (for example in planning of public transportation we would like to minimize the number of transfers between individual means of transport).

Finally, let us note that another attempt to define formalism for describing planning problems is represented by *Abstract Description Language (ADL)* (Pednault, 1989). The ADL represents a tradeoff between the expressiveness of general logical formulas and the computational requirements. Most of the features of ADL were adopted by the PDDL language.

HOLES problems. These planning problems encode the classical form of Dirichlet's box principle. That is, we have n holes and $n+1$ pigeons; the task is to place all the pigeons into hole such that no two pigeons are in the same hole. The problem is insolvable. The description of the set of planning operators is as follows in the PDDL language.

```
(define (domain holes)
  (:requirements :strips :typing)
  (:types type)
  (:predicates (empty ?hole) (out ?pigeon) (in ?pigeon ?hole)
               (placed ?pigeon))
  (:action fill
    :parameters (?hole ?pigeon)
    :precondition (and (empty ?hole)
                       (out ?pigeon)
                  )
    :effect  (and (in ?pigeon ?hole)
                  (placed ?pigeon)
                  (not (out ?pigeon))
                  (not (empty ?hole))))
  )
)
```

The description of the initial state and the description of the goal for 4 holes and 5 pigeons are as follows in the PDDL language.

```
(define (problem holes-05_04)
  (:domain holes)
  (:objects p1 p2 p3 p4 p5
            h1 h2 h3 h4)
  (:init
   (out p1) (out p2) (out p3) (out p4) (out p5)
   (empty h1) (empty h2) (empty h3) (empty h4)
  )
  (:goal (and (placed p1) (placed p2) (placed p3)
              (placed p4) (placed p5)))
)
```

●

JAM problems. This set of problems again encodes the Dirichlet's box principle. However, now the whole problem is solvable. The Dirichlet's box principle represents a bottleneck in the problem (problems are called according to this bottleneck - jam, according to the traffic jam). To find a step optimal plan for the problem it is necessary to detect in-

solvability of the box principle which makes the problem difficult.

More precisely, the problem consists of a Dirichlet's box principle followed by a further processing. We have $n+1$ different pigeons that must go through the set of n holes (no two pigeons can be in the same hole - unsolvable in a single time-step but solvable in two time-steps) before they can be further processed.

```
(define (domain jam)
  (:requirements :strips :typing)
  (:types type)
  (:predicates (empty ?hole) (out ?pigeon) (in ?pigeon ?hole)
               (placed ?pigeon) (color ?pigeon ?color)
               (next ?color1 ?color2))
  (:action fill
    :parameters (?hole ?pigeon)
    :precondition (and (empty ?hole)
                       (out ?pigeon)
                   )
    :effect  (and (in ?pigeon ?hole)
                  (placed ?pigeon)
                  (not (out ?pigeon))
                  (not (empty ?hole))
             )
  )
  (:action switch
    :parameters (?pigeon ?color1 ?color2)
    :precondition (and (color ?pigeon ?color1)
                       (placed ?pigeon)
                       (next ?color1 ?color2)
                   )
    :effect  (and (color ?pigeon ?color2)
                  (not (color ?pigeon ?color1))
       )
  )
  (:action leave
    :parameters (?hole ?pigeon)
    :precondition (and (in ?pigeon ?hole)
                       (placed ?pigeon))
    :effect  (and (out ?pigeon)
                  (empty ?hole)
                  (not (in ?pigeon ?hole))
                  (not (placed ?pigeon)))
  )
)
```

The following PDDL code describes the initial state and the goal for the problem consisting of 4 holes and 5 pigeons.

```
(define (problem jam-05_04)
  (:domain jam)
  (:objects p1 p2 p3 p4 p5 h1 h2 h3 h4 red blue)
  (:init
   (next red blue)
```

```
   (out p1) (out p2)  (out p3) (out p4) (out p5)
   (empty h1) (empty h2) (empty h3) (empty h4)
   (color p1 red) (color p2 red) (color p3 red)
   (color p4 red) (color p5 red)
  )
  (:goal (and (out p1) (out p2) (out p3) (out p4) (out p5)
              (color p1 blue) (color p2 blue) (color p3 blue)
              (color p4 blue) (color p5 blue)
         )
  )
)
```

●

UJAM problems. This set of problems represents unsolvable version of the problem from the previous *jam* problem. Again the problem encodes Dirichlet's box principle. The principle is now present in two stages where the first stage is solvable while the second stage is unsolvable. Both principles represent a bottleneck on the number of steps.

More precisely, we have a set o $n+1$ pigeons and two sets of holes. Both sets consist of n holes. The pigeons must go through the holes from the first stage before they can be further processed (the principle is unsolvable in a single time-step but solvable in two time-steps). After this processing the pigeons can continue to their final positions in the second set of n holes (this principle is unsolvable).

```
(define (domain ujam)
  (:requirements :strips :typing)
  (:types type)
  (:predicates (empty ?hole) (out ?pigeon) (in ?pigeon ?hole)
               (placed ?pigeon) (picked ?pigeon) (remaining ?pick)
               (color ?pigeon ?color) (next ?color1 ?color2)
               (visible ?color))
  (:action fill
    :parameters (?hole ?pigeon)
    :precondition (and (empty ?hole)
                       (out ?pigeon)
                  )
    :effect  (and (in ?pigeon ?hole)
                  (placed ?pigeon)
                  (not (out ?pigeon))
                  (not (empty ?hole))
             )
  )
  (:action switch
    :parameters (?pigeon ?color1 ?color2)
    :precondition (and (color ?pigeon ?color1)
                       (placed ?pigeon)
                       (next ?color1 ?color2)
                   )
    :effect  (and (color ?pigeon ?color2)
                  (not (color ?pigeon ?color1))
             )
  )
```

```
  (:action leave
    :parameters (?hole ?pigeon)
    :precondition (and (in ?pigeon ?hole)
                       (placed ?pigeon))
    :effect  (and (out ?pigeon)
                  (empty ?hole)
                  (not (in ?pigeon ?hole))
                  (not (placed ?pigeon))
             )
  )
  (:action pick
    :parameters (?pick ?pigeon ?color)
    :precondition (and (remaining ?pick)
                       (out ?pigeon)
                       (visible ?color)
                       (color ?pigeon ?color)
                  )
    :effect  (and (picked ?pigeon)
                  (not (out ?pigeon))
                  (not (remaining ?pick))
             )
  )
)
```

The following PDDL code represents the problem consisting of 5 pigeons and 4+4 holes.

```
(define (problem ujam-05_04)
  (:domain ujam)
  (:objects p1 p2 p3 p4 p5 h1 h2 h3 h4 s1 s2 s3 s4 red blue)
  (:init
   (next red blue) (visible blue)
   (out p1) (out p2)  (out p3) (out p4)  (out p5)
   (color p1 red) (color p2 red) (color p3 red) (color p4 red)
   (color p5 red)
   (remaining s1) (remaining s2) (remaining s3) (remaining s4)
   (empty h1) (empty h2) (empty h3) (empty h4)
  )
  (:goal (and (picked p1) (picked p2) (picked p3)
              (picked p4) (picked p5)
         )
  )
)
```

●

APPENDIX B

REMOVABLE MEDIUM

The thesis includes a removable medium (DVD-ROM) with additional material in the electronic form. The medium contains electronic version of the text of the thesis, the source code of software used to produce the experimental results, the raw experimental data, and additional tables with results that did not fit into the text of the thesis. The content of the attached medium is summarized in the table B.1.

Content of the attached removable medium		
Directory		*Brief description*
plan		***Source code*** and ***results*** concerning ***classical planning*** with planning graphs.
	plan/experiments	***Raw experimental*** data for planning problems. The results use a xml-like format. Statistical characteristics are available in text form.
	plan/problems	***Planning problems*** described in a xml-like format. Problems of three planning domains from chapter 3 are listed here.
	plan/splan	***Source code*** of the software used to produce experimental data in planning (chapter 3). The directory contains three versions of the ***SPlan*** - experimental planning system.
sat		***Source code*** and ***problems*** concerning ***Boolean satisfiability***
	sat/problems	***Boolean satisfaction problems.***
	sat/ssat	***Source code*** of the software used to produce experimental data in Boolean satisfiability (chapter 4). The directory contains ***SSat*** - Boolean satisfaction problem experimental preprocessor.
tables		***Additional tables*** and ***graphs*** with results that did not fit in the text of the thesis.
text		***Text of the thesis*** in the electronic form.
tools		***Auxiliary software tools*** used for research of the thesis topic.
	tools/ge	***Source code*** of the ***Graph Explorer*** - experimental visualization tool for graphs.
	tools/graphs	Several ***problems*** concerning planning and Boolean satisfiability ***reinterpreted as graphs***. Graphs can be viewed in Graph Explorer.

Table B.1. *CONTENT OF THE ATTACHED REMOVABLE MEDIUM.* A brief summary of the content of the attached removable medium.